VISIT…

I N S T R U C T O R ' S M A N U A L

f o r

A n I n t r o d u c t i o n

t o

D a t a b a s e S y s t e m s

——— ♦♦♦♦♦ ———

Eighth Edition

by

C. J. Date

Copyright (c) 2003 C. J. Date

General Remarks

The purpose of this manual is to give guidance on how to use the eighth edition of the book *An Introduction to Database Systems*——referred to throughout the manual as simply "the book," or "this book," or "the present book," or just "the eighth edition"——as a basis for teaching a database course. The book is suitable for a primary (one- or two-semester) course at the junior or senior undergraduate or first-year graduate level; it also contains some more forward-looking and research-oriented material that would be relevant to a more advanced course. Students are expected to have a basic understanding of (a) the storage and file management capabilities (indexing, hashing, etc.) of a modern computer system, and (b) the features of a typical high-level programming language (Java, Pascal, C, PL/I, etc.).

Let me immediately say a little more regarding these two prerequisites:

1. In connection with the first, please note that although the book proper contains nothing on the subject, there's an online appendix available——Appendix D, "Storage Structures and Access Methods——that does provide a tutorial overview of such matters. That appendix is an upgraded version of material that *was* included in the book proper in the first six editions. But file management isn't specific to database systems; what's more, it's a huge subject in its own right, and it has textbooks of its own——see, e.g., *File Organization for Database Design,* by Gio Wiederhold, published by McGraw-Hill in 1987 (which, despite the title, is really about files, not databases). That's why I've dropped the inline coverage of such material from the last two editions of the present book.

2. In connection with the second, please note that the book uses a hypothetical language called **Tutorial D** as a basis for examples throughout. **Tutorial D** might be characterized, loosely, as a Pascal-like language; it's defined in detail in reference [3.3]. (See the subsection immediately following for an explanation of this reference format. I'll have more to say regarding reference [3.3] in particular later in these introductory notes——see the subsection on *The Third Manifesto,* pages 6-8.)

All of that being said, I want to say too that I don't think either of these prerequisites is particularly demanding; but you should be prepared, as an instructor, to sidetrack occasionally

Copyright (c) 2003 C. J. Date

and give a brief explanation of (e.g.) what indexes are all about, if the question arises.

A note on style: The book itself follows convention in being written in the first person plural (we, our, etc.). This manual, by contrast, is written in the first person singular (I, my, etc.)——except where (a) it quotes directly from the book, or (b) it reflects ideas, opinions, positions, etc., that are due to both Hugh Darwen and myself (again, see the subsection on ***The Third Manifesto,*** pages 6-8). The latter case applies particularly to Chapter 20 on type inheritance, Chapter 23 on temporal databases, and Chapter 26 on object/relational databases.

The manual is also a little chattier than the book, using elisions such as "it's" and "they're" instead of the more stilted "it is" and "they are," etc.

Structure of the Book

The book overall consists of a preface plus 27 chapters (divided into six parts), together with four appendixes, as follows:

Part I : Preliminaries

1. An Overview of Database Management
2. Database System Architecture
3. An Introduction to Relational Databases
4. An Introduction to SQL

Part II : The Relational Model

5. Types
6. Relations
7. Relational Algebra
8. Relational Calculus
9. Integrity
10. Views

Part III : Database Design

11. Functional Dependencies
12. Further Normalization I: 1NF, 2NF, 3NF, BCNF
13. Further Normalization II: Higher Normal Forms
14. Semantic Modeling

Part IV : Transaction Management

15. Recovery
16. Concurrency

Part V : Further Topics

17. Security
18. Optimization
19. Missing Information
20. Type Inheritance
21. Distributed Databases
22. Decision Support
23. Temporal Databases
24. Logic-Based Databases

Part VI : Objects, Relations, and XML

25. Object Databases
26. Object/Relational Databases
27. The World Wide Web and XML

Appendixes

A. The TransRelational[tm] Model
B. SQL Expressions
C. Abbreviations, Acronyms, and Symbols
D. Storage Structures and Access Methods *(online only)*

The preface gives more specifics regarding the contents of each part, chapter, etc. It also summarizes the major differences between this eighth edition and its immediate predecessor.

By the way, if you're familiar with earlier editions, I'd like to stress the point that this edition, like each of its predecessors, is in large degree a brand new book——not least because (of course) I keep learning myself and improving my own understanding, and producing a new edition allows me to correct past mistakes. (In this connection, I'd like to draw your attention to the wonderful quote from Bertrand Russell in the book's preface. Also please note the epigraphs by George Santayana and Maurice Wilkes! It would be nice if the computer science community would take these remarks to heart.)

The following notes, also from the book's preface, are lightly edited here:

(Begin quote)

The book overall is meant to be read in sequence more or less as written, but you can skip later chapters, and later sections within chapters, if you choose. A suggested plan for a first reading would be:

- Read Chapters 1 and 2 "once over lightly."

- Read Chapters 3 and 4 very carefully.

- Read Chapters 5, 6, 7, 9, and 10 carefully, but skip Chapter 8——except, probably, for Section 8.6 on SQL (in fact, you might want to treat portions of Section 8.6 "early," perhaps along with the discussion of embedded SQL in Chapter 4). ***Note:*** It would be possible to skip or skim Chapter 5, too, but if you do you'll need to come back and deal with it properly before you cover Chapter 20 or Chapters 25-27.

- Read Chapter 11 "once over lightly."

- Read Chapters 12 and 14 carefully, but skip Chapter 13. (You could also read Chapter 14 earlier if you like, possibly right after Chapter 4. Many instructors like to treat the entity/relationship material much earlier than I do. For that reason I've tried to make Chapter 14 more or less self-contained, so that it can be read "early" if you like.)

- Read Chapters 15 and 16 carefully.

- Read subsequent chapters selectively (but in sequence), according to taste and interest.

I'd like to add that instructors, at least, should read the preface too (most people don't!).

Each chapter opens with an introduction and closes with a summary; each chapter also includes a set of exercises (and the online answers often give additional information about the subject at hand). Each chapter also includes a set of references, many of them annotated. This structure allows the subject matter to be treated in a multi-level fashion, with the most important concepts and results being presented inline in the main body of the text and various subsidiary issues and more complex aspects being deferred to the exercises, or answers, or reference annotation, as appropriate.

With regard to those references, by the way, I should explain that references are identified in the text by two-part numbers in square brackets. For example, the reference "[3.1]" refers to the first item in the list of references at the end of Chapter 3: namely, a paper by E. F. Codd published in ***CACM 25,*** No. 2, in February, 1982. (For an explanation of abbreviations used in references——e.g., "CACM"——see Appendix B. Regarding Codd in particular, let me draw your attention to the dedication in this new edition of the book. It's a sad comment on the state of our field that I often encounter database students or professionals who have never heard of Ted Codd.)

(End quote)

This manual gives more specific guidance, with rationale, on what can safely be skipped and what really ought not to be. As indicated above, it also gives answers to the exercises——or most of them, at any rate; note, however, that some exercises don't have any single "right" answer, but instead are intended to promote group discussion and perhaps serve as some kind of miniproject. Such cases are flagged in this manual by the phrase ***No answer provided. Note:*** The book also includes a number of inline exercises embedded in the body of the text, and the remarks of this paragraph apply to those inline exercises too.

Structure of this Manual

The broad structure of this manual mirrors that of the book itself: It consists of this preface, together with notes on each part, each chapter, and each appendix from the subject book (including the online Appendix D). Among other things, the notes on a given part or chapter or appendix:

- Spell out what that piece of the book is trying to achieve
- Explain the place of that piece in the overall scheme of things
- Describe and hit the highlights from the relevant text
- Indicate which items can be omitted if desired and which must definitely not be
- Include additional answers to exercises (as already noted)

and, more generally, give what I hope are helpful hints regarding the teaching of the material.

The Third Manifesto

You might be aware that, along with my colleague Hugh Darwen, I published another database book a little while back called ***The Third Manifesto*** [3.3].* ***The Third Manifesto*** consists of a detailed technical proposal for the future of data and database systems; not surprisingly, therefore, the ideas contained therein inform the present book throughout. Which isn't to say ***The Third Manifesto*** is a prerequisite to the present book——it isn't; but it ***is*** directly relevant to much that's in this book, and further pertinent information is often to be found there. Instructors in

particular really ought to have a copy available, if only for reference purposes. (I realize this recommendation is somewhat self-serving, but I make it in good faith.) Students, on the other hand——at least beginning students——would probably find much of *The Third Manifesto* pretty heavy going. It's more of a graduate text, not an undergraduate one.

———————

* The full title is *Foundation for Future Database Systems: The Third Manifesto* (2nd edition, Addison-Wesley, 2000). The first edition (1998) had the slightly different title *Foundation for Object/Relational Databases: The Third Manifesto;* however, it wasn't exclusively about object/relational databases as such, which was why we changed the title for the second edition. By the way, there's a website, too: *http://www.thethirdmanifesto.com*. The website *http://www.dbdebunk.com* also contains much relevant material.

———————

I should explain why we called that book *The **Third** Manifesto*. The reason is that there were two previous ones:

- *The Object-Oriented Database System Manifesto* [20.2,25.1]
- *The Third Generation Database System Manifesto* [26.44]

Like our own *Manifesto,* each of these documents proposes a basis for future DBMSs. However:

- The first essentially ignores the relational model! In our opinion, this flaw is more than enough to rule it out immediately as a serious contender.
- The second does agree that the relational model mustn't be ignored——but unfortunately goes on to say that supporting the relational model means supporting *SQL*.

The Third Manifesto, by contrast, takes the position that any attempt to move forward, if it's to stand the test of time, must reject SQL unequivocally (see the next subsection, "Some Remarks on SQL," for further elaboration of this point). Of course, we're not so stupid as to think SQL is going to go away; after all, COBOL has never gone away. *Au contraire,* SQL databases and SQL applications are obviously going to be with us for a long time to come. So we do have to worry about what to do about today's "SQL legacy," and *The Third Manifesto* does include some specific

suggestions in this regard. Further discussion of those suggestions would be out of place here, however.

The Third Manifesto also discusses and stresses several important **logical differences** (the term is due to Wittgenstein)——i.e., differences that are quite simple, yet crucial, and ones that many people (not to mention products!) seem to get confused over. Some of the differences in question are:

- Model *vs.* implementation
- Value *vs.* variable
- Type *vs.* representation
- Read-only operator *vs.* update operator
- Argument *vs.* parameter
- Type *vs.* relation

and so on (this isn't meant to be an exhaustive list). These notes aren't the place to spell out exactly what all of the differences are (in any case, anyone who claims to be an instructor in this field should be thoroughly familiar with them already); rather, my purpose in mentioning them here is to alert you to the fact that they are appealed to numerous times throughout the book, and also to suggest that you might want to be on the lookout for confusion over them among your students. Of course, the various differences *are* all explained in detail in *The Third Manifesto,* as well as in the book itself.

As noted earlier, *The Third Manifesto* also includes a definition of **Tutorial D**——although, to be frank, there shouldn't be any need to refer to that definition in the context of the present book (the **Tutorial D** examples should all be pretty much self-explanatory).

Some Remarks on SQL

As noted in the previous subsection, *The Third Manifesto* takes the position that any attempt to move forward, if it's to stand the test of time, **must reject SQL**. This rather heretical position clearly needs some defending; after all, earlier editions of *An Introduction to Database Systems* actually used SQL to illustrate relational ideas, in the belief that it's easier on the student to show the concrete before the abstract. Unfortunately, however, the gulf between SQL and the relational model has now grown so

wide that I feel it would be actively misleading to continue to use it for such a purpose. Indeed, we're talking here about another huge logical difference: SQL and the relational model aren't the same thing!——and in my opinion it's categorically *not* a good idea (any more) to use SQL as a vehicle for teaching relational concepts. *Note:* I make this observation in full knowledge of the fact that many database texts and courses do exactly what I'm here saying they shouldn't.

At the risk of beating a dead horse, I'd like to add that SQL today is, sadly, so far from being a true embodiment of relational principles——it suffers from so many sins of both omission and commission (see, e.g., references [4.15-4.20] and [4.22])——that my own preference would have been to relegate it to an appendix, or even to drop it entirely. However, SQL is so important from a commercial point of view (and every database professional does need to have some familiarity with it) that it really wouldn't have been appropriate to dismiss it in so cavalier a fashion. I've therefore settled on a compromise: a chapter on SQL basics in Part I of the book (Chapter 4), and individual sections in later chapters describing those aspects of SQL, if any, that are relevant to the subject of the chapter in question. (You can get some idea of the extent of that SQL coverage from the fact that there are "SQL Facilities" sections in 14 out of a total of 23 subsequent chapters.)

The net result of the foregoing is that, while the eighth edition does in fact discuss all of the most important aspects of SQL, the language overall is treated as a kind of second-class citizen. And while I feel this treatment is appropriate for a book of the kind the eighth edition is meant to be, I do recognize that some students need more emphasis on SQL specifically. For such students, I believe the book provides the basics——not to mention the proper solid theoretical foundation——but instructors will probably need to provide additional examples etc. of their own to supplement what's in the book. (In this connection, I'd like, somewhat immodestly, to recommend reference [4.20] as a good resource.)

What Makes this Book Different?

The following remarks are also taken from the book's preface, but again are lightly edited here:

(Begin quote)

Every database book on the market has its own individual strengths and weaknesses, and every writer has his or her own particular ax to grind. One concentrates on transaction management issues; another stresses entity/relationship modeling; another looks at

everything through a SQL lens; yet another takes a pure "object" point of view; still another views the field exclusively in terms of commercial products; and so on. And, of course, I'm no exception to this rule——I too have an ax to grind: what might be called the **foundation** ax. I believe very firmly that we must get the foundation right, and understand it properly, before we try to build on that foundation. This belief on my part explains the heavy emphasis in this book on the relational model; in particular, it explains the length of Part II——the most important part of the book——where I present my own understanding of the relational model as carefully as I can. I'm interested in foundations, not fads and fashions. Products change all the time, but principles endure.

In this regard, I'd like to draw your attention to the fact that there are several important "foundation" topics for which this book, virtually alone among the competition, includes an entire in-depth chapter (or an appendix, in one case). The topics in question include:

- Types
- Integrity
- Views
- Missing information
- Inheritance
- Temporal databases
- The TransRelational Model

In connection with that same point (the importance of foundations), I have to admit that the overall tone of the book has changed over the years. The first few editions were mostly descriptive in nature; they described the field as it actually was in practice, "warts and all." Later editions, by contrast, were much more ***pre***scriptive; they talked about the way the field ***ought*** to be and the way it ought to develop in the future, if we did things right. And the eighth edition is certainly prescriptive in this sense (in other words, it's a text with an attitude!). Since the first part of that "doing things right" is surely educating oneself as to what those right things actually are, I hope this new edition can help in that endeavor.

(End quote)

The foregoing remarks explain (among other things) the comparative lack of emphasis on SQL. Of course, it's true that students who learn the theory thoroughly first are going to have a few unpleasant surprises in store if and when they get out into the commercial world and have to deal with SQL products; it's also

true that tradeoffs and compromises sometimes have to be made in a commercial context. However, I believe very firmly that:

- Any such tradeoffs and compromises should always be made *from a position of conceptual strength*.

- Such tradeoffs and compromises should *not* have to be made in the academic or research world.

- An emphasis on the way things ought to be, instead of on the way things currently are, makes it a little more likely that matters will improve in the future.

So the focus of the book is clearly on theory. But that doesn't mean it's not practical! It's my *very strong opinion* that the theory we're talking about is very practical indeed, and moreover that products that were faithful to that theory would be *more* practical——certainly more user-friendly, and probably easier to implement——than the products that are currently out there in the commercial world.

And one more point: When I say the focus is on theory, I mean, primarily, that the focus is on the **insights** such theory can provide. The book contains comparatively little in the way of formal proofs and the like——such material can always be found in the research literature, and appropriate references to that literature are included in the book. Rather, the emphasis throughout is on *insight* and *understanding* (and precision), not so much on formalisms. And I believe it's this emphasis that truly sets the book apart from the competition.

Concluding Remarks

The field of database management has grown very large, and it can be divided up in various ways. One clear division is into model *vs*. implementation issues, and (as should be clear from what I've already said above) the book's focus is very heavily on the former rather than the latter. However, please don't interpret this fact as meaning that I think implementation issues are unimportant——of course not! But I do think we should know **what** we're trying to do, and **why**, before getting into the specifics of **how**. Thus, I believe implementers too should be familiar with the material covered in the book. (I also believe that "data model"* people should have some knowledge of implementation issues, but for present purposes I regard that as a separate and secondary point. Though the book certainly doesn't ignore implementation issues entirely! In this connection, see in particular Chapter 18 and Appendixes A and D.)

---

* See Chapter 1 for a discussion of the two very different meanings of the term *data model*. I'm using it here in its primary——i.e., more fundamental and more important——sense.

To repeat, the field has grown very large, a fact that accounts for the book's somewhat embarrassing length. When I wrote the first edition, I tried to be comprehensive; now, with this new edition, I can claim only that the book is, as advertised, truly an *introduction* to the subject. Accordingly, I've tried to concentrate on topics that genuinely are fundamental and primary (the relational model being the obvious example), and I've gone into less detail on matters that seem to me to be secondary (decision support might be an example here).

This brings me to the end of these introductory notes. Let me close by wishing you well in your attempts to teach this material, and indeed in all of your database activities. If you have any comments or questions, I can be reached via the publisher, Addison Wesley Longman, at 75 Arlington St. #300, Boston, Mass. 02116, care of Katherine Harutunian (617/848-7518). Thank you for your interest.

Healdsburg, California C. J. Date
2003

***** End of Preface *****

P A R T I

P R E L I M I N A R I E S

The introduction to Part I in the book itself is more or less self-explanatory:

(Begin quote)

Part I consists of four introductory chapters:

- Chapter 1 sets the scene by explaining what a database is and why database systems are desirable. It also briefly discusses the differences between relational systems and others.

- Next, Chapter 2 presents a general architecture for database systems, the so-called ANSI/SPARC architecture. That architecture serves as a framework on which the rest of the book will build.

- Chapter 3 then presents an overview of relational systems (the aim is to serve as a gentle introduction to the much more comprehensive discussions of the same subject in Part II and later parts of the book). It also introduces and explains the running example, the suppliers-and-parts database.

- Finally, Chapter 4 introduces the standard relational language SQL (more precisely, SQL:1999).

(End quote)

Chapters 1 and 2 can probably be covered quite quickly. Chapter 3 must be treated thoroughly, however, and the same almost certainly goes for Chapter 4 as well. See the notes on the individual chapters for further elaboration of these remarks.

***** End of Introduction to Part I *****

Copyright (c) 2003 C. J. Date

Chapter 1

An Overview of Database Management

Principal Sections

- What's a DB system?
- What's a DB?
- Why DB?
- Data independence
- Relational systems and others

General Remarks

The purpose of Chapter 1 is, of course, basically just to set the scene for the chapters to follow; it covers a lot of "necessary evils" that have to be addressed before we can get on to the interesting stuff. In a live course, however, I doubt whether it's necessary (or even desirable) to spend too much time on this material up front. As the chapter itself says at the end of Section 1.1 (the following quote is slightly edited here):

(Begin quote)

While a full understanding of this chapter and the next is necessary to a proper appreciation of the features and capabilities of a modern database system, it can't be denied that the material is somewhat abstract and rather dry in places (also, it does tend to involve a large number of concepts and terms that might be new to you). In Chapters 3 and 4 you'll find material that's much less abstract and hence more immediately understandable, perhaps. You might therefore prefer just to give these first two chapters a "once over lightly" reading for now, and to reread them more carefully later as they become more directly relevant to the topics at hand.

(End quote)

The fact is, given the widespread availability and use of database systems today (on desktop and laptop computers in particular), many people have a basic understanding of what a database is already. In order to motivate the students,

Copyright (c) 2003 C. J. Date

therefore, it might be sufficient just to give a brief discussion of an example such as the following. *Note:* This particular example is due to Roger King. It was used in the instructor's manual for earlier editions. Of course, you can use a different example if you like, but please note that Roger's example illustrates (in particular) the distinction between a database system as such and a **file** system, and any replacement example should do likewise.

(Begin example)

Before the age of database systems, data-intensive computer systems often involved a maze of separate files. Consider an insurance company, for example. One division might be processing claims, and there might be many thousands of such claims every day. Another might be keeping track of hundreds of thousands of subscriber accounts, processing premium payments and maintaining personal data. The actuarial division might be maintaining statistics on the relative risks of various kinds of subscribers. The underwriting division might be developing group insurance plans and calculating appropriate premium charges. You can see that the actuaries need access to claim data in order to calculate their statistics, the underwriters need access to subscriber information for obvious reasons, the claims personnel need access to underwriting data and subscriber information in order to know who is covered and how, and so on.

As this example suggests, a large company maintains massive amounts of data, and its various employees must share that data, and share it simultaneously. In fact, the example illustrates the two key properties of a database system: Such a system must allow the enterprise (a) to **integrate** its data and (b) to **share** that integrated data effectively.

(End example)

To repeat, examples like the foregoing might suffice by way of motivation, and much of the chapter might thus be skipped on a first reading. For this reason, it's really not worth giving a blow-by-blow analysis of the individual sections here. However, some attention should certainly be paid to the concept of **(physical) data independence** and the associated distinctions between:

a. **Logical** *vs.* **physical** issues

b. **Model** *vs.* **implementation** issues

In connection with the second of these distinctions (which is really a special case of the first), the general concept of a **data**

model should also be covered (with illustrations of objects and operators taken specifically from the discussion of the **relational** model earlier in the chapter). Further:

- Stress the fact that the term ***data model*** is used in the database field with two different meanings, but we'll be using it almost exclusively in the more general and more important——in fact, more fundamental——sense.

- Also stress the point that most existing database systems are based on the relational model; most future database systems are likely to be so, too; and hence the emphasis throughout the book is on relational systems (for good solid ***theoretical*** as well as practical reasons!). Hugh Darwen's article [1.2] is strongly recommended, for instructors as well as students; ideally, it should be provided as a handout.

- Also stress the point that this book concentrates on model issues rather than implementation ones. Both kinds of issues are important, of course, but——in the industrial world, at least, and to some extent in the academic world as well——model issues tend to take a back seat to implementation ones (in fact, most people have only a hazy understanding of the distinction). It's my position that while the model folks obviously need the implementation folks, the opposite is true too (possibly "even more true"), and yet isn't nearly as widely appreciated. To repeat a remark from the preface to this manual (at least in essence): It's important to know **what** we're trying to do, and **why,** before getting into the specifics of **how**.

  Other items that mustn't be omitted at this early stage:

- What **SQL** is (of course), with simple examples of SELECT, INSERT, UPDATE, and DELETE——with the emphasis on ***simple,*** however. One reviewer of a previous edition objected to the fact that simple SQL coding examples and exercises appeared in this introductory chapter before SQL is discussed in depth. Here's my response to that criticism:

  a. The SQL **examples** are included in order to give the "flavor" of database languages, and in particular to illustrate the point that such languages typically include statements to perform the four basic operations of ***data retrieval, insertion, deletion,*** and ***replacement***. They're meant to be (and in fact are) pretty much self-explanatory!

  b. As for the SQL **exercises,** it seems to me that very little extrapolation from the examples is required on the part of the student in order to understand the exercises and answer

them (they really aren't very difficult). Of course, the exercises can be skipped if desired.

- **Terminology:** Terminology is such a problem ... We often have several terms for the same thing (or almost the same thing), and this point raises its ugly head almost immediately. To be specific, explain (a) files / records / fields *vs.* (b) tables / rows / columns *vs.* (c) relations / tuples / attributes.

- **Data types:** Stress the point that data types are *not* limited to simple things like numbers and strings.

- **Entities and relationships:** Stress that a relationship is really just a special kind of entity. Distinguish carefully between entity *types* and entity *occurrences* (or *instances*). Myself, I wouldn't elide either "type" or "occurrence"——in this context or any other——until the concepts have become second nature to the students (and maybe not even then). Perhaps mention that the relational model in particular represents both entities and relationships in the same way, which is one of the many reasons why it's simpler and more flexible than other models.

- Simple **E/R diagrams** (if you like; most people do like these things, though I don't much myself): If you do cover them here, at least mention that they fail to capture the most important part of any design, *viz.,* integrity constraints! See the further remarks on this subject in this manual in the notes on Chapter 14, especially in the annotation to reference [14.39].

- Explain the basic concept of a **transaction**. (Also, be aware that Chapter 16 offers some heretical opinions on this topic——but don't mention those opinions here, of course.)

- Explain the basic concepts of **security** and **integrity**. *Note:* These concepts are often confused; be sure to distinguish between them properly! The following rather glib definitions from Chapter 17 might help:

  a. *Security* means making sure users are **allowed** to do the things they're trying to do.

  b. *Integrity* means making sure the things they're trying to do are **correct**. (By the way: Don't get into this issue right now, but this question of correctness is a tricky one. We'll be taking a much closer look at it in Chapters 9 and 16.)

- Introduce the basic concept of a **relation**——and explain that (a) *relation* is not the same as *relationship* and (b) relations don't contain *pointers*.

  A couple more points for the instructor:

a. If you mention network database systems at all, you might want to warn the students that "network" in this context refers to a certain *data structure,* not to a data communications network like the Internet!

b. A nice but perhaps rather sophisticated way to think about a relational system is the following: Such a system consists of a *relational language compiler* together with a *very extensive run-time system*. (I wouldn't mention this point unless asked about it, but it might help understanding for some more "advanced" students.)

Finally, let me call your attention to a couple of small points: the double underlining convention for primary key columns in figures (as in, e.g., Fig. 1.1), and the preferred (and official) pronunciation "ess-cue-ell" for SQL.

Answers to Exercises

1.1 Some of the following definitions elaborate slightly on those given in the book *per se*.

- A **binary relationship** *type* is a relationship type involving exactly two entity types (not necessarily distinct). Analogously, of course, a binary relationship *instance* is a relationship instance involving exactly two entity instances (again, not necessarily distinct). *Note:* As an example of a binary relationship instance in which the two entity instances aren't distinct, consider the often heard remark to the effect that so-and-so is his or her own worst enemy!

- A **command-driven interface** is an interface that permits the user to issue requests to the system by means of explicit *commands* (also known as *statements*), typically expressed in the form of text strings in some formal language such as SQL.

- **Concurrent access** means——at least from the user's point of view——that several users are allowed to use the same DBMS (more precisely, the same copy or instance of the same DBMS) to access the same database at the same time. The system provides controls to ensure that such concurrent access does not cause incorrect results (at least in principle; however, see further discussion in Chapter 16).

- **Data administration** is the task of (a) deciding what data should be kept in the database and (b) establishing the necessary policies for maintaining and dealing with that data once it has been entered into that database.

- A **database** is a repository for a collection of computerized data files. (At least, this would be the normal definition. A much better definition is: A database is a collection of propositions, assumed by convention to be ones that evaluate to TRUE. See reference [1.2] for further explanation.)

- A **database system** is a computerized system whose overall purpose is to maintain a database and to make the information in that database available on demand. (As in the body of the chapter, we assume for simplicity, here and throughout these answers, that all of the data in the system is in fact kept in just one database. This assumption is very unrealistic in practice.)

- *(Physical)* **data independence** is the immunity of applications to changes in storage structure (how the data is physically stored) and access technique (how it is physically accessed). *Note: Logical* data independence is discussed in Chapters 2, 3, and especially 10. See also Appendixes A and D.

- The **database administrator** (DBA) is the person whose job it is to create the actual database and to implement the technical controls needed to enforce the various policy decisions made by the data administrator. The DBA is also responsible for ensuring that the system operates with adequate performance and for providing a variety of other related technical services.

- The **database management system** (DBMS) is a software component that manages the database and shields users from low-level details (in particular, details of how the database is physically stored and accessed). All requests from users for access to the database are handled by the DBMS.

  Caveat: Care is needed over terminology here. The three concepts *database, DBMS product,* and *DBMS instance* are (obviously) logically distinct. Yet the term *DBMS* is often used to mean either *DBMS product* or *DBMS instance,* as the context demands, and the term *database* is often used to mean *DBMS* in either sense. What's more, the term *DBMS* is even used on occasion to mean the database! In the book and this manual the unqualified term *database* ALWAYS means database, not DBMS,

and the unqualified term ***DBMS*** ALWAYS means DBMS instance, not DBMS product.

- An **entity** is any distinguishable person, place, or thing that is deemed to be of interest for some reason. Entities can be as concrete or as abstract as we please. A **relationship** (*q.v.*) is a special kind of entity. (As with relationships, we really need to distinguish between entity ***types*** and entity ***occurrences*** or ***instances,*** but in informal contexts the same term ***entity*** is often used for both concepts.)

- An **entity/relationship diagram** is a pictorial representation of (a) the entities (more accurately, entity ***types***) that are of interest to some enterprise and (b) the relationships (more accurately, relationship ***types***) that hold among those entities. ***Note:*** The point is worth making that while an E/R diagram ***might*** represent "all" of the entities of interest, it is virtually certain that it will not represent all of the ***relationships*** of interest. The fact is, the term "relationship" in an E/R context really refers to a very special kind of relationship——***viz.,*** the kind that is represented in a relational database by a foreign key. But foreign key relationships are far from being the only possible ones, or the only ones that might be of interest, or even the most important ones.

- A **forms-driven interface** is an interface that permits the user to issue requests to the system by filling in "forms" on the screen (where the term "form" refers to an on-screen analog of some conventional paper form).

- **Integration** means the database can be thought of as a unification of several otherwise distinct data files, with any redundancy among those files wholly or partly eliminated.

- **Integrity** means, loosely, ***accuracy*** or ***correctness;*** thus, the problem of integrity is the problem of ensuring——insofar as is possible——that the data in the database does not contain any incorrect information. ***Note:*** The integrity concept is CRUCIAL and FUNDAMENTAL, as later chapters (especially Chapter 9) make clear.

- A **menu-driven interface** is an interface that permits the user to issue requests to the system by selecting and combining items from predefined menus displayed on the screen.

- A **multi-user system** is a system that supports concurrent access (*q.v.*). It is contrasted with a **single**-user system.

- An **online application** is an application whose purpose is to support an end user who is accessing the database from an online workstation or terminal.

- **Persistent data** is data whose lifetime typically exceeds that of individual application program executions. In other words, it is data that (a) is stored in the database and (b) persists from the moment it is created until the moment it is explicitly destroyed. (*Non*persistent data, by contrast, is typically destroyed implicitly when the application program that created it ceases execution, or possibly even sooner.)

- A **property** is some characteristic or feature possessed by some entity (or some relationship). Examples are a person's name, a part's weight, a car's color, or a contract's duration. (By the way, is a contract an entity or a relationship? What do you think? Justify your answer!)

- A **query language** is a language that supports the expression of high-level commands (such as SELECT, INSERT, etc.) to the DBMS. SQL is an example of such a language. *Note:* Despite the name, query languages typically support much more than just query——i.e., retrieval——operations alone. (Though not always! OQL and XQuery——see Chapter 25 and Chapter 27, respectively——are examples of query languages that do support retrieval only.)

- **Redundancy** means the very same piece of information (say the fact that a certain employee is in a certain department) is recorded more than once, possibly in more than one way. Note that redundancy at the physical storage level is often desirable (for performance reasons), while redundancy at the logical user level is usually undesirable (because it complicates the user interface, among other things). But physical redundancy need not imply logical redundancy, so long as the system provides an adequate degree of data independence.

- A **relationship** is an association among entities. *Note:* As with entities, it is strictly necessary to distinguish between relationship *types* and relationship *occurrences* or *instances,* but in informal contexts we often use the same term *relationship* for both concepts.

- **Security** means the protection of the data in the database against unauthorized access.

- **Sharing** refers to the possibility that individual pieces of data in the database can be shared among several different

users, in the sense that each of those users can have access to the same piece of data, possibly even at the same time (and different users can use it for different purposes).

- A **stored field** is the smallest unit of stored data.* The type *vs.* occurrence (or instance) distinction is important once again, just as it is with entities and relationships.

---

* But see Appendix A (regarding not only this term but also the terms *stored file* and *stored record*).

- A **stored file** is the collection of all currently existing occurrences of one type of stored record.

- A **stored record** is a collection of related stored fields. The type *vs.* occurrence distinction is important yet again.

- A **transaction** is a logical unit of work, typically involving several database operations (in particular, several update operations), whose execution is guaranteed to be *atomic*——i.e., all or nothing——from a logical point of view.

1.2 Some of the *advantages* are as follows:

- Compactness
- Speed
- Less drudgery
- Currency
- Centralized control
- Data independence

Some of the *disadvantages* are as follows:

- Security might be compromised (without good controls).
- Integrity might be compromised (without good controls).

Copyright (c) 2003 C. J. Date

- Additional hardware might be required.

- Performance overhead might be significant.

- Successful operation is crucial (the enterprise might be highly vulnerable to failure).

- The system is likely to be complex (though such complexity should be concealed from the user).

1.3 A **relational system** is a system that is based on the relational model. Loosely speaking, therefore, it is a system in which:

a. The data is perceived by the user as tables (and nothing but tables).

b. The operators at the user's disposal (e.g., for data retrieval) are operators that generate new tables from old.

In a nonrelational system, by contrast, the user is presented with data in the form of other structures, either instead of or in addition to the tables of a relational system. Those other structures, in turn, require other operators to manipulate them. For example, in a *hierarchic* system, the data is presented to the user in the form of a set of tree structures (hierarchies), and the operators provided for manipulating such structures include operators for traversing hierarchic paths——in effect, *following pointers*——up and down those trees.

Note: It's worth pointing out that, in a sense, a relation might be thought of as a special case of a hierarchy (to be specific, it's a *root-only* hierarchy). In principle, therefore, a hierarchic system requires all of the relational operators *plus* certain additional operators. And those additional operators certainly add complexity, but they don't add any functionality (there's nothing useful that can be done with hierarchies that can't be done with just relations).

1.4 A **data model** is an abstract, self-contained, logical definition of the objects,* operators, and so forth, that together constitute the abstract machine with which users interact (the objects allow us to model the *structure* of data, the operators allow us to model its *behavior*). An **implementation** of a given data model is a physical realization on a real machine of the components of that model. In a nutshell: The model is what users have to know about; the implementation is what users don't have to know about.

Copyright (c) 2003 C. J. Date

---

* The term ***object*** is being used here in its generic sense, not its special object-oriented sense.

The difference between model and implementation is important because (among other things) it forms the basis for achieving data independence.

1.5 a.

WINE	PRODUCER
Zinfandel	Rafanelli

b.

WINE	PRODUCER
Chardonnay	Buena Vista
Chardonnay	Geyser Peak
Joh. Riesling	Jekel
Fumé Blanc	Ch. St. Jean
Gewurztraminer	Ch. St. Jean

c.

BIN#	WINE	YEAR
6	Chardonnay	2002
22	Fumé Blanc	2000
52	Pinot Noir	1999

d.

WINE	BIN#	YEAR
Cab. Sauvignon	48	1997

1.6 We give a solution for part a. only: "Rafanelli is a producer of Zinfandel"——or, more precisely, "Some bin contains some bottles of Zinfandel that were produced by Rafanelli in some year, and they will be ready to drink in some year."

1.7 a. The specified row (for bin number 80) is added to the CELLAR table.

Copyright (c) 2003 C. J. Date

b. The rows for bin numbers 45, 48, 64, and 72 are deleted from the CELLAR table.

c. The row for bin number 50 has the number of bottles set to 5.

d. Same as c.

Incidentally, note how convenient it is to be able to refer to rows by their primary key value (the primary key for the CELLAR table is {BIN#}——see Chapter 8). In other words, such key values effectively provide a ***row-level addressing mechanism*** in a relational system.

1.8 a.
```
SELECT BIN#, WINE, BOTTLES
FROM   CELLAR
WHERE  PRODUCER = 'Geyser Peak' ;
```

b.
```
SELECT BIN#, WINE
FROM   CELLAR
WHERE  BOTTLES > 5 ;
```

c.
```
SELECT BIN#
FROM   CELLAR
WHERE  WINE = 'Cab. Sauvignon'
OR     WINE = 'Pinot Noir'
OR     WINE = 'Zinfandel'
OR     WINE = 'Syrah'
OR     ..... ;
```

There's no shortcut answer to this question, because "color of wine" isn't explicitly recorded in the database; thus, the DBMS doesn't know that (e.g.) Pinot Noir is red.

d.
```
UPDATE CELLAR
SET    BOTTLES = BOTTLES + 3
WHERE  BIN# = 30 ;
```

e.
```
DELETE
FROM   CELLAR
WHERE  WINE = 'Chardonnay' ;
```

f.
```
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 55, 'Merlot', 'Gary Farrell', 2000, 12, 2005 ) ;
```

1.9 *No answer provided.*

***** End of Chapter 1 *****

Chapter 2

Database System Architecture

Principal Sections

- The three levels of the architecture
- The external level
- The conceptual level
- The internal level
- Mappings
- The DBA
- The DBMS
- Data communications
- Client/server architecture
- Utilities
- Distributed processing

General Remarks

This chapter resembles Chapter 1 in that it's probably best given just a "once over lightly" treatment on a first pass. As with Chapter 1, therefore, it's not really worth giving a blow-by-blow analysis of the individual sections here. However, the following topics, at least, should be touched on in a live class:

- The **external, conceptual,** and **internal** levels (and common synonyms——e.g., *physical* or *stored* in place of internal, *community logical* or just *logical* in place of conceptual, *user logical* or just *logical* in place of external ... the terminology issue rears its ugly head again!).

- **DDLs, DMLs,** and **schemas** (the last of these also known more simply as *data definitions*).

- Point out that the relational model has **nothing** explicit to say regarding the internal level (deliberately, of course).

- **Logical data independence** (at least a brief mention, with a forward reference to Chapters 3 and——especially——10).

- Steps in **processing and executing a DML request** (hence, an overview of the basic components of a DBMS).

Copyright (c) 2003 C. J. Date

- Basic **client/server** concepts (and note that client *vs.* server is, primarily, a *logical* distinction, not a physical one).

- Basic idea (very superficial) of **distributed systems**.

Note: Section 2.2 and (to a lesser extent) subsequent sections make use of a rather trivial example based on PL/I and COBOL. Of course, I do realize that PL/I and COBOL are regarded as antediluvian in some circles (though they're still very significant commercially), but which actual languages are used isn't important! What's more, no PL/I- or COBOL-specific knowledge is really needed in order to follow the example. Naturally you can substitute your own favorite more modern languages if you prefer.

Answers to Exercises

2.1 See Fig. 2.3 in the body of the chapter.

2.2 Some of the following definitions elaborate slightly on those given in the body of the chapter.

- **Back end:** Same as *server, q.v.*

- A **client** is an application that runs on top of the DBMS——either a user-written application or a "built-in" application, i.e., an application provided by the DBMS vendor or some third-party software vendor. The term is also used to refer to the hardware platform the client application runs on, especially when that platform is distinct from the one the server runs on.

- The **conceptual view** is an abstract representation of the database in its entirety. The **conceptual schema** is a definition of that conceptual view. The **conceptual DDL** is a language for writing conceptual schemas.

- The **conceptual/internal mapping** defines the correspondence between the conceptual view and the stored database.

- A **data definition language** (DDL) is a language for defining, or *declaring,* database objects.

- The **data dictionary** is a system database that contains "data about the data"——i.e., *definitions* of other objects in the system, also known as *metadata* (in particular, all of the various schemas and mappings will physically be stored, in

both source and object form, in the dictionary). A comprehensive dictionary will also include cross-reference information, showing, for instance, which applications use which pieces of the database, which users require which reports, what terminals or workstations are connected to the system, and so on. The dictionary might even——in fact, probably should——be integrated into the database it defines, and thus include its own definition (i.e., be "self-describing").

- A **data manipulation language** (DML) is a language for "manipulating" or processing database objects.

- A **data sublanguage** is that portion of a given language that's concerned specifically with database objects and operations. It might or might not be clearly separable from the host language (*q.v.*) in which it's embedded or from which it's invoked.

- A **database/data-communications system** (DB/DC system) is a combination of a DC manager and a DBMS, in which the DBMS looks after the database and the DC manager handles all messages to and from the DBMS (or, more accurately, to and from applications that use the DBMS).

- The **data communications manager** (DC manager) is a software component that manages all message transmissions between the user and the DBMS (more accurately, between the user and some application running on top of the DBMS).

- A **distributed database** is (loosely) a database that is logically centralized but physically distributed across many distinct physical sites. It's a little difficult to make this definition more precise (different writers tend to use the term in different ways); carried to its logical conclusion, however, full support for distributed database implies that a single application should be able to operate "transparently" on data that is spread across a variety of different databases, managed by a variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected together by a variety of different communication networks——where "transparently" means that the application operates from a logical point of view as if the data were all managed by a single DBMS running on a single machine.

- **Distributed processing** means that distinct machines can be connected together into some kind of communications network, in such a way that a single data processing task can be spread

across several machines in the network (and, typically, carried out in parallel).

- An **external view** is a more or less abstract representation of some portion of the total database. An **external schema** is a definition of such an external view. An **external DDL** is a language for writing external schemas.

- An **external/conceptual mapping** defines the correspondence between an external view and the conceptual view.

- **Front end:** Same as *client, q.v.*

- A **host language** is a language in which a data sublanguage is embedded. The host language is responsible for providing various nondatabase facilities, such as I/O operations, local variables, computational operations, if-then-else logic, and so on.

- **Load** is the process of creating the initial version of the database (or portions thereof) from one or more nondatabase files.

- **Logical database design** is the process of identifying the entities of interest to the enterprise and identifying the information to be recorded about those entities. *Note:* Chapter 9 and Part III of the book make it clear that *integrity constraints* are highly relevant to the logical database design process. Note too that logical design should be done *before* the corresponding physical design (*q.v.*).

- The **internal view** is the database as physically stored.* The **internal schema** is the definition of that internal view. The **internal DDL** is a language for writing internal schemas. *Note:* The book usually uses the more intuitive terms "stored database" and "stored database definition" in place of "internal view" and "internal schema," respectively.

---

* A slight oversimplification. To paraphrase some remarks from Section 2.5, the internal view is really "at one remove" from the physical level, since it doesn't deal with *physical* records——also called blocks or pages——nor with device-specific considerations such as cylinder or track sizes. In other words, it effectively assumes an unbounded linear address space; details of how that address space maps to physical storage are highly system-specific and are deliberately omitted from the general architecture.

- **Physical database design** is the process of deciding how the logical database design is to be physically represented at the stored database level.

- A **planned request** is a request for which the need was foreseen well in advance of the time at which the request is actually to be executed. The DBA will probably have tuned the physical database design in such a way as to guarantee good performance for planned requests.

- **Reorganization** is the process of rearranging the way the data is stored at the physical level. It is usually (perhaps always, in the last analysis) done for performance reasons.

- The **server** is the DBMS *per se*. The term is also used to refer to the hardware platform the DBMS runs on, especially when that platform is distinct from the one the clients run on.

- **Stored database definition:** Same as *internal schema, q.v.*

- **Unload/reload** is the process of unloading the database, or portions thereof, to backup storage for recovery purposes and subsequently reloading the database (or portions thereof) from such backup copies. *Note:* Load and reload are usually done by means of the same utility, of course.

- An **unplanned request** is an *ad hoc* query, i.e., a request for which the need wasn't seen in advance, but instead arose in a spur-of-the-moment fashion.

- The **user interface** is essentially just the system as seen by the user. In other words, it's essentially identical to an *external view,* in the ANSI/SPARC sense.

- A **utility** is a program designed to help the DBA with some administration task, such as load or reorganization.

2.3 As explained in the body of the chapter, any given external record occurrence will require fields from several conceptual record occurrences (in general), and each conceptual record occurrence in turn will require fields from several stored record occurrences (in general). Conceptually, then, the DBMS must first retrieve all required stored record occurrences; next, construct the required conceptual record occurrences; finally, construct the

required external record occurrence. At each stage, data type or other conversions might be necessary.

2.4 The major functions performed by the DBMS include:

- Data definition support
- Data manipulation support
- Data security and integrity support
- Data recovery and concurrency support
- Data dictionary support

Of course, it's desirable that the DBMS perform all of these functions as efficiently as possible.

2.5 Logical data independence means users and user programs are immune to changes in the logical structure of the database (meaning changes at the conceptual or "community logical" level). **Physical** data independence means users and user programs are immune to changes in the physical structure of the database (meaning changes at the internal or stored level). A good DBMS will provide both.

2.6 Metadata or **descriptor data** is "data about the data"——i.e., ***definitions*** of other objects in the system. Examples include all of the various schemas and mappings (external, conceptual, etc.) and all of the various security and integrity constraints. Metadata is kept in the **dictionary** or **catalog**.

2.7 The major functions performed by the DBA include:

- Defining the conceptual schema (i.e., logical database design; done in conjunction with the data administrator)
- Defining the internal schema (i.e., physical database design)
- Liaising with users (help write the external schemas, etc.)
- Defining security and integrity constraints
- Defining backup and recovery procedures
- Monitoring performance and responding to changing requirements

This isn't an exhaustive list.

Copyright (c) 2003 C. J. Date

2.8 The **file manager** is that component of the overall system that manages stored files (it's "closer to the disk" than the DBMS is). It supports the creation and destruction of stored files and simple retrieval and update operations on stored records in such files. In contrast to the DBMS, the typical file manager:

- Is unaware of the internal structure of stored records, and hence can't handle requests that rely on a knowledge of that structure
- Provides little or no security or integrity support
- Provides little or no recovery or concurrency support
- Doesn't support a true data dictionary
- Provides much less data independence

In addition, files are typically not "integrated" or "shared" in the same sense that the database is, but instead are usually private to some particular user or application. See Appendix D for further discussion.

2.9 Such tools fall into many categories:

- Query language processors
- Report writers
- Business graphics subsystems
- Spreadsheets
- Natural language processors
- Statistical packages
- Copy management or data extract tools
- Application generators (including 4GL processors)
- Other application development tools, including computer-aided software engineering (CASE) products
- Data mining and visualization tools

Copyright (c) 2003 C. J. Date

and so on. Specific commercial examples are beyond the scope of this text (any database trade publication will include references to any number of such products).

2.10 Examples of database utilities include:

- Load routines
- Unload/reload routines
- Reorganization routines
- Statistical routines
- Analysis routines

and many others.

2.11 *No answer provided.*

***** End of Chapter 2 *****

Chapter 3

An Introduction to Relational Databases

Principal Sections

- An informal look at the relational model
- Relations and relvars
- What relations mean
- Optimization
- The catalog
- Base relvars and views
- Transactions
- The suppliers-and-parts DB

General Remarks

The overall purpose of this chapter is to give the student "the big picture" of what database systems (in particular, relational systems) are and how they work. It thus provides a framework for the more detailed information presented in later chapters to build on. *The chapter is therefore crucial, at least for students who are new to database technology; it mustn't be skipped, skimped, or skimmed* (except possibly as indicated below).

3.2 An Informal Look at the Relational Model

Briefly discuss **structural, integrity,** and **manipulative aspects** and **restrict, project,** and **join** operations. Mention **types** (and explain the "domain" terminology). Stress the relational **closure** property and the **set-at-a-time** nature of relational operations. Cover **The Information Principle,*** and in particular its "no pointers" corollary (no pointers visible to the user, that is). Mention **primary** and **foreign keys** (but *don't* discuss them in depth). Explain who Ted Codd is (or was, rather; sadly, Ted died as this book was going to press).

Copyright (c) 2003 C. J. Date

* *The Information Principle,* along with several other important principles to be discussed in later chapters, is repeated at the back of the book (overleaf from the left endpaper).

Note: The book favors the more formal term *restrict* over the possibly more common name *select* in order to avoid confusion with the SELECT operator of SQL.

The section closes with a rather terse abstract definition of the relational model. Don't attempt to explain that definition at this point, but mention that we'll come back to it later (at the very end of Chapter 10).

3.3 Relations and Relvars

The following analogy is helpful in explaining the basic point of this section. Suppose we say in some programming language:

```
DECLARE N INTEGER ... ;
```

N here is **not** an integer; it's an integer *variable* whose *values* are integers *per se*——different integers at different times (that's what *variable* means). In exactly the same way, if we say in SQL:

```
CREATE TABLE T ... ;
```

T here is **not** a table (or, as I'd prefer to say, *relation*)——it's a relation (table) *variable* whose *values* are relations (tables) *per se*——different relations (tables) at different times.* Thus, when we "update T" (e.g., by "inserting a row"), what we're really doing is *replacing the old relation value of T* en bloc *by a new, different relation value*. Of course, it's true that the old value and the new value are somewhat similar——the new one just has one more row than the old one——but conceptually they *are* different values. (In mathematics, the sets {*a,b,c*} and {*a,b,c,d*} are *different sets*——there's no notion of one somehow being just an "updated version" of the other.)

---

* T can be regarded as a *relation* variable rather than a table variable only if various SQL quirks are ignored and not "taken advantage of." In particular, there must be no duplicate rows,

there must be no nulls, and we must ignore the left-to-right column ordering.

———————

The term **relvar** (= relation variable) is not in common usage but ought to be!——much confusion has arisen over the years from the fact that the same term, ***relation*** (***table,*** in SQL contexts), has been used for these two very different concepts:

- ***Relations*** are **values;** they can thus be "read" but not updated, by definition. (The one thing you can't do to ***any*** value is update it——for if you could, then after such an update it wouldn't be the same value any more. E.g., consider the value that's the integer 3.)

- ***Relvars*** are **variables;** they can thus be "read" **and** updated, by definition. (In fact, "variable" really ***means*** "updatable." To say that something is a variable is to say, precisely, that that something can be used as the target of an assignment operation——no more and no less.)

The unqualified term "relation" is thus short for ***relation value,*** just as, e.g., the unqualified term "integer" is short for ***integer value***.

Note: The distinction between values and variables in general is a crucial one, and both instructors and students should be very clear on it. It's a distinction that permeates the entire computing field, the entire database field, and the entire book. (It's worth mentioning too in passing that the object world tends to be somewhat confused over it!) See Chapter 1 of ***The Third Manifesto*** or the answer to Exercise 5.2 in this manual for further elaboration.

Observe now that the operations of the relational algebra all apply to ***relations*** (possibly to the relations that happen to be the current values of relvars), not to relvars as such; the only operation that applies to relvars specifically is (relational) **assignment,** together with its shorthand forms INSERT, DELETE, and UPDATE. Observe too that ***update operations*** and ***integrity constraints*** both apply specifically to relvars, not relations.

The book uses **Tutorial D** instead of SQL to explain concepts, for reasons explained in the preface (Section 3.3 is the first place in the book in which **Tutorial D** syntax appears). This fact should not cause any difficulties——**Tutorial D** is a "Pascal-like" language and should be easy enough to follow for any reader having the prerequisites stated in the preface.

Copyright (c) 2003 C. J. Date

By the way, now that we know about relvars, we have another way of stating *The Information Principle:* The only variables allowed in a relational database are, specifically, relvars.

3.4 What Relations Mean

Regarding the business of users being able to define their own **types,** give a forward reference to Chapter 5. This functionality wasn't included in SQL:1992 but is part——the major new part, in fact——of SQL:1999, and we'll be looking at it in detail when we get to Chapter 5.

The concepts **heading, body, predicate,** and **proposition** are all *ABSOLUTELY FUNDAMENTAL*. Note that they apply to relation variables as well as relation values. Stress the point that propositions in general aren't necessarily true ones, but those represented by rows in relational tables are assumed (or believed) to be so. Perhaps mention the *Closed World Assumption* or *Interpretation* (covered in more detail in Chapter 6).

Note: There's a possible source of confusion here. Sometimes we put rows in the database whose truth we're not certain of (loosely speaking); thus it might be felt that we can't say that "all rows in the database correspond to true propositions." If this issue comes up, explain that it's taken care of either via the predicate ("it's true that we are fairly sure but not definite that such and such is true") or via an explicit "confidence factor" column ("it's true that our confidence level that such and such is true is *x* percent").

Emphasize the point that *every* relation, base or derived, has a predicate. Ditto relvars.

Types and relations are (a) necessary, (b) sufficient, (c) not the same thing!

3.5 Optimization

Don't go into too much detail; simply show (by example) the increased simplicity in query formulation that **automatic navigation** affords, and explain that the **optimizer** has to do some "smart thinking" in order to support such automatic navigation. Forward references to Chapters 7 and 18.

Note: This section of the book includes the following example of a relational expression, expressed (of course) in **Tutorial D:**

```
( EMP WHERE EMP# = EMP# ('E4') ) { SALARY }
```

Observe:

- The use of braces surrounding the commalist of names of columns over which the projection is to be done (in the example, of course, that commalist contains just one name). **Tutorial D** generally uses braces when the enclosed material is supposed to represent a ***set*** of items, as here. ***Note:*** See Section 4.6 in the book or the next chapter in this manual for an explanation of the term "commalist."

- The EMP# literal (actually a ***selector invocation***) EMP#('E4'). Don't get into details here: Just say that this expression denotes a specific employee number, and we'll be talking about such things in detail in Chapter 5. (In fact, other EMP# literals also appeared in other examples earlier in the chapter.)

3.6 The Catalog

The catalog was mentioned in Chapter 2. Here just stress the point that the catalog in a relational system will itself consist of relvars——of course!

The section closes with the following inline exercise: "What does the following do?"

```
( ( TABLE JOIN COLUMN )
          WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

Answer: This relational expression (or "query") yields table- and column-name pairs for tables with fewer than five columns.

3.7 Base Relvars and Views

One reason it's desirable to explain the basic notion of views at this early stage in the book is so that we can distinguish base relvars from them!——and hence explain base relvars, and go on to distinguish such relvars from "stored" ones. (The notion of "base" relvars can't be properly explained if there isn't any other kind.) Introducing views here as another kind of relvar also serves as a little subtle softening up for the discussion of ***The Principle of Interchangeability*** in Chapter 10.

Views are (named) derived relvars——and, conceptually at least, they're **virtual,** i.e., not materialized. Of course, it's true that some systems do ***implement*** views via materialization, but

that's an implementation matter, not part of the model. It's also true that more recently some systems (typically data warehouse systems) have started talking about "materialized views" (see Chapters 10 and 22), but that's a model *vs.* implementation confusion! Such "materialized views" are better called *snapshots* (they aren't really views at all, and *snapshot* was the original term for the concept in question). Snapshots are discussed in Chapter 10.

Operations on views are translated, at least conceptually, via **substitution** into operations on the underlying data. Thus, views provide *logical data independence*.

Do *not* fall into:

- The trap of equating base and stored relvars
- The trap of taking the term "tables" (or "relations" or "relvars") to mean, specifically, *base* tables (or relations or relvars) only

People fall into both of these traps all too often, especially in SQL contexts. The SQL standard, for example, makes frequent use of expressions such as "tables and views"——implying very strongly that a view isn't a table. And yet *the whole point* about a view is that it *is* a table (much as, in mathematics, the whole point about a subset is that it is a set). To fall into either of these traps is to fail to think relationally. And this failure leads to mistakes: mistakes in databases, mistakes in applications, mistakes in the design of SQL itself.

3.8 Transactions

The usual stuff here (the topic is not peculiar to relational systems): **BEGIN TRANSACTION, COMMIT, ROLLBACK;** atomicity, durability, isolation, serializability. (Incidentally, note that these are *not* exactly "the ACID properties"; that's deliberate, and so is the lack of reference to the ACID acronym.) Superficial!——this is just an introduction. Forward references to Chapters 15 and 16.

3.9 The Suppliers-and-Parts DB

More or less self-explanatory. Note the user-defined types (forward reference to Chapter 5). As the summary section says (more or less): "It's worth taking the time to familiarize yourself with this example now, if you haven't already done so; that is, you should at least know which relvars have which columns

and what the primary and foreign keys are (it isn't so important to know exactly what the sample data values are!)." Mention the fact that Fig. 3.8 is repeated inside the back cover of the book, for ease of subsequent reference.

Answers to Exercises

3.1 As usual, some of the following definitions elaborate slightly on those given in the body of the chapter.

- The term **automatic navigation** refers to the fact that (in a relational system) the process of "navigating" around the stored data in order to implement user requests is performed automatically by the system, not manually by the user.

- A **base relvar**——also known as a *real* relvar [3.3]——is a relvar that has *independent* or *autonomous existence*. More precisely, it's a relvar that isn't a derived relvar (*q.v.*). It's *not* necessarily the same thing as a "stored relvar."

- The **catalog** is a set of system relvars whose purpose is to contain *descriptors* regarding the various objects that are of interest to the system itself, such as base relvars, views, indexes, users, integrity constraints, security constraints, and so on.

- The term **closure** (of relational operations) refers to the fact that (a) the output from any relational operation is the same kind of object as the input——they're all relations——and so (b) the output from one operation can become input to another. Closure implies that we can write *nested (relation-valued) expressions*.

  Note: We stress the point that when we say that the output from each operation is another relation, we are talking *from a conceptual point of view*. We don't necessarily mean to imply that the system actually has to materialize the result of every individual operation in its entirety. In fact, of course, the system tries very hard not to, if such materialization is logically unnecessary (see the brief discussion of *pipelined evaluation* in the body of the chapter).

- **Commit** is the operation that signals successful end-of-transaction. Any updates made to the database by the transaction in question are now "made permanent" and become visible to other transactions.

- A **derived relvar** is a relvar whose value at any given time is the result of evaluating a specified relational expression, typically involving other relvars (ultimately, *base* relvars). Note that (like a base relvar) a derived relvar is still a variable![*]——in other words, the term "relvar" does *not* refer just to base relvars; moreover, derived relvars must be updatable (for otherwise they cannot be said to be variables).

_____

[*] To be more precise, a derived relvar is a variable if and only if its defining relational expression involves at least one relvar; otherwise it would be more accurate to think of it as a relation *constant* (a "relcon"?), and it wouldn't be updatable.

_____

- A **foreign key** is a column or combination of columns in one relvar whose values are required to match those of the *primary* key in some other relvar (or possibly in the same relvar). *Note:* This definition is only approximate. A more precise definition is given in Chapter 9 (where, among other things, the point is stressed that a foreign key is a *set* of columns and a foreign key value is a *set* of values——in fact, a (sub)tuple).

- **Join** is a relational operation that joins two relations together on the basis of common values in a common column. *Note:* This definition is only approximate. A more precise definition is given in Chapter 7.

- **Optimization** is the process of deciding how to implement user access requests. In other words, it's the process of deciding how to perform *automatic navigation* (*q.v.*).

- A **predicate** is a truth-valued function. Every relation has a corresponding predicate that defines (loosely) "what the relation means." Each row in a given relation denotes a certain **true proposition,** obtained from the predicate by substituting certain argument values of the appropriate type for the parameters of the predicate ("instantiating the predicate"). *Note:* These remarks are all true of relvars as well as relations, *mutatis mutandis*.

- The **primary key** of a given relvar is a column or combination of columns in that relvar whose values can be used to identify rows within that relvar uniquely (in other words, it's a

unique identifier for the rows of that relvar). *Note:* This definition is only approximate. A more precise definition is given in Chapter 9 (where, among other things, the point is stressed that a primary key is a *set* of columns and a primary key value is a *set* of values——in fact, a (sub)tuple).

- **Projection** is a relational operation that extracts specified columns from a relation. *Note:* This definition is only approximate. A more precise definition is given in Chapter 7.

- A **proposition** is, loosely, something that evaluates to either TRUE or FALSE, unequivocally.

- A **relational database** is a database in which the data is perceived by the user at any given time as relations (and nothing but relations). Equivalently, a relational database is *a container for relvars* (and nothing but relvars).

- A **relational DBMS** is a DBMS that supports relational databases and relational operations such as restrict, project, and join on the data in those databases.

- **The relational model** is an abstract theory of data that's based on certain aspects of mathematics (principally set theory and predicate logic). It can be thought of as *a way of looking at data*——i.e., as a prescription for a way of representing data (namely, by means of relations), and a prescription for a way of manipulating such a representation (namely, by means of operators such as join). *Note:* The very abstract definition of the relational model given at the end of Section 3.2 is explained in detail in Chapter 10 of these notes (in the answer to Exercise 10.20).

- **Restriction** (also known as *selection*) is a relational operation that extracts specified rows from a relation. *Note:* This definition is only approximate. A more precise definition is given in Chapter 7.

- **Rollback** is the operation that signals unsuccessful end-of-transaction. Any updates made to the database by the transaction in question are "rolled back" (undone) and are never made visible to other transactions.

- A **set-level operation** is an operation that operates on entire sets as operands and returns an entire set as a result. Relational operations are all set-level, since they operate on and return entire relations, and relations contain *sets* of rows.

- A (relational) **view**——also known as a ***virtual relvar*** [3.3]——is a named derived relvar. Views are ***virtual,*** in the sense that they don't have any existence apart from the base relvars from which they're derived (but users should typically not be aware that a given view is in fact virtual in this sense, though SQL falls very short in this regard, owing to its weak support for view updating). Operations on views are processed by translating them into equivalent operations on those underlying base relvars.

3.2 The following figure doesn't include the catalog entries for relvars TABLE and COLUMN themselves. ***Note:*** The figure is incomplete in many other ways as well. See Exercise 5.10 in Chapter 5.

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
S	4	5	.....
P	5	6	.....
SP	3	12	.....
.....	.....	.....	.....

COLUMNS

TABNAME	COLNAME	.....
S	S#	.....
S	SNAME	.....
S	STATUS	.....
S	CITY	.....
P	P#	.....
P	PNAME	.....
P	COLOR	.....
P	WEIGHT	.....
P	CITY	.....
SP	S#	.....
SP	P#	.....
SP	QTY	.....
.....	.....	.....

3.3 The following figure shows the entries for the TABLE and COLUMN relvars only (i.e., the entries for the user's own relvars are omitted). It's obviously not possible to give precise COLCOUNT and ROWCOUNT values.

TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
TABLES	*(>3)*	*(>2)*	.....
COLUMNS	*(>2)*	*(>5)*	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
TABLE	TABNAME	.....
TABLE	COLCOUNT	.....
TABLE	ROWCOUNT	.....
COLUMN	TABNAME	.....
COLUMN	COLNAME	.....
.....	.....	.....

3.4 The query retrieves supplier number and city for suppliers who supply part P2.

3.5 The meaning of the query is "Get supplier numbers for London suppliers who supply part P2." The first step in processing the query is to replace the name V by the expression that defines V, giving:

```
( ( ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY } )
                                  WHERE CITY = 'London' ) { S# }
```

This simplifies to:

```
( ( S WHERE CITY = 'London' ) JOIN
                           ( SP WHERE P# = P# ('P2') ) ) { S# }
```

For further discussion and explanation, see Chapters 10 and 18.

3.6 Atomicity means that transactions are guaranteed (from a logical point of view) either to execute in their entirety or not to execute at all, even if (say) the system fails halfway through the process. **Durability** means that once a transaction successfully commits, its updates are guaranteed to be applied to the database, even if the system subsequently fails at any point. **Isolation** means that database updates made by a given transaction *T1* are kept hidden from all distinct transactions *T2* until and unless *T1* successfully commits. **Serializability** means that the interleaved execution of a set of concurrent transactions is guaranteed to produce the same result as executing those same transactions one at a time in some (unspecified) serial order.

Copyright (c) 2003 C. J. Date

3.7 *The Information Principle* states that the entire information content of the database is represented in one and only one way: namely, as explicit values in column positions in rows in tables. Equivalently: The database contains relvars, and nothing but relvars. *Note:* As indicated in the chapter, *The Information Principle* might better be called *The Principle of Uniform Representation*.

3.8 *No answer provided*.

***** End of Chapter 3 *****

Chapter 4

An Introduction to SQL

Principal Sections

- Overview
- The catalog
- Views
- Transactions
- Embedded SQL
- Dynamic SQL and SQL/CLI
- SQL isn't perfect

General Remarks

The overall purpose of Chapter 3 was to give the student the big picture of what relational systems in general are (or should be!) all about. By contrast, the overall purpose of the present chapter is to give the student the big picture of what SQL systems in particular are all about.

All SQL discussions in the book are based on the current standard SQL:1999 (except for a few brief mentions here and there of the expected next version, SQL:2003). Warn the students that "their mileage may vary" when it comes to commercial SQL dialects!——see reference [4.22]. Also warn them that we deliberately won't be using SQL as a vehicle for teaching database principles; we'll cover the principles *first* and then consider how (and to what extent) those principles are realized——or departed from——in SQL *afterward*. While SQL is obviously important from a pragmatic standpoint, it's a very poor realization of proper database principles, as well as being a very poorly designed language from just about any standpoint. Better that students learn proper concepts and principles first before getting their heads bent out of shape by SQL.

Incidentally, I can't resist the temptation to point out that it's really a bit of a joke——or a confidence trick——to be talking about "SQL:2003," when nobody has yet implemented even *SQL:1992* in its entirety, let alone SQL:1999. Nor in fact could anybody do so!——given that SQL:1992 is full of gaps and contradictions, gaps and contradictions that still exist in SQL:1999 and will certainly still exist in SQL:2003 as well. See reference [4.20], Appendix

Copyright (c) 2003 C. J. Date

D, for an extended discussion of some of those gaps and contradictions.

I also can't resist mentioning the fact that upgrading the SQL coverage to the SQL:1999 level caused me more trouble than anything else in producing the eighth edition. The 1999 standard is simultaneously enormous in size and extremely hard to understand (in this regard, you can get a sense of the general flavor from the not atypical quote that appears in Chapter 10, Section 10.6; that same quote is repeated in Chapter 10 of this manual).

The foregoing negative remarks notwithstanding, the chapter ***per se*** contains little in the way of detailed or specific criticism; rather, such criticisms appear, where relevant, at appropriate points in later chapters. See also references [4.15-4.20] at the end of the chapter. ***Note:*** The chapter and all "SQL Facilities" sections in later chapters could be skipped if the course is concerned only with principles and not pragma. But few instructors are likely to enjoy such a luxury.

One point instructors need to be aware of: Exercise 4.1 introduces the extended version of the running suppliers-and-parts example (***viz.,*** suppliers, parts, and projects). Subsequent chapters tend to use suppliers-and-parts as a basis for the main body of the text and suppliers, parts, and projects as a basis for exercises; however, this separation is **not** rigidly adhered to. Be aware, therefore, that there might be some occasional potential for confusion in this area. The endpapers can help here (Figs. 3.8 and 4.5 are both repeated inside the back cover).

BNF Notation

Chapter 4 is the first in the book to use standard BNF notation, or rather a simple variant thereof. The variant in question——which isn't explained in detail in the book——is defined as follows:

- Special characters and material in uppercase must be written exactly as shown. Material in lowercase enclosed in angle brackets "<" and ">" represents a syntactic category that appears on the left side of another production rule, and hence must eventually be replaced by specific items chosen by the user.

- Vertical bars "|" are used to separate alternatives.

- Square brackets "[" and "]" are used to indicate that the material enclosed in those brackets is optional.

The text also makes extensive use of a shorthand based on **lists** and **commalists**. These terms *are* explained in the book (in Sections 5.4 and 4.6, respectively), but I'll repeat the explanations here for convenience. Let *<xyz>* denote an arbitrary syntactic category (i.e., anything that appears on the left side of some BNF production rule). Then:

- The expression *<xyz list>* denotes a sequence of zero or more *<xyz>*s in which each pair of adjacent *<xyz>*s is separated by one or more blanks.

- The expression *<xyz commalist>* denotes a sequence of zero or more *<xyz>*s in which each pair of adjacent *<xyz>*s is separated by a comma (and possibly one or more blanks on either side of the comma).

Give some simple examples.

4.2 Overview

SQL talks in terms of **tables** (and rows and columns), not *relations* (and tuples and attributes). SQL is often said to include both **data definition** and **data manipulation** facilities (though these terms have become increasingly inappropriate as SQL has expanded to become a computationally complete programming language*). It also includes a bunch of miscellaneous other facilities.

---

* With the ratification of SQL/PSM in 1996, SQL is indeed now computationally complete——entire applications can now be written in SQL, without any need for a distinct host language (except for I/O facilities, which SQL doesn't provide).

Regarding *data definition,* cover **CREATE TABLE** and (briefly) built-in scalar types. *Note:* User-defined types were added in SQL:1999, and we'll discuss them in detail in the next chapter (we'll say a bit more in that chapter about built-in types as well). Do *not* discuss SQL-style "domains"! (See reference [4.20] for an explanation of how SQL-style domains differ from true types.)

Regarding *data manipulation,* cover **SELECT** (including "SELECT *" and SELECT formulations of restrict, project, and join queries) and *set-level* **INSERT, DELETE,** and **UPDATE** (no relational assignment

as such!). Note carefully, however, that this section deliberately doesn't get into a lot of detail on SELECT (and so the exercises and answers don't, either); such matters are deferred to Section 8.6, after the relevant relational concepts have been described.* INSERT, DELETE, and UPDATE, by contrast, are *not* explained much further in any later chapter (the treatment here is it, more or less).

---

* If you like, you could beef up the treatment of SELECT by bringing in some of the material from Section 8.6 in here.

4.3 The Catalog / 4.4 Views / 4.5 Transactions

Briefly survey the relevant SQL features:

- **Information Schemas**.
- **CREATE VIEW** and the substitution mechanism (how does it look in SQL?——leads to a brief introduction to **nested subqueries**). Do *not* get into details of SQL view updating.
- **START TRANSACTION, COMMIT WORK, ROLLBACK WORK**. No need to get into the effect of these operations on cursors yet (unless anyone asks)——that material's covered in Chapter 15. Don't mention SET TRANSACTION. *Note:* START TRANSACTION was added in SQL:1999; prior to that, transactions could be started in SQL only implicitly, a state of affairs that caused some grief. For reasons of backward compatibility, of course, it's still possible to start transactions implicitly, but I wouldn't get into this unless anyone asks about it. A tiny point of syntax: It's really odd that the SQL committee chose to call the operator START TRANSACTION and not BEGIN TRANSACTION, given that BEGIN was already a reserved word and START wasn't. An illustration of the point that designing a language by committee isn't a very good idea?

4.6 Embedded SQL

This section is probably the most important in the chapter; it gives details (some of them unfortunately a little tedious) that don't logically belong anywhere else in the book. Discuss:

- **The dual-mode principle**

- **SQLSTATE**

- **Singleton SELECT, INSERT,** and **searched DELETE and UPDATE**

- **Cursors** (in reasonable detail, including **DECLARE CURSOR, ORDER BY, OPEN, CLOSE, FETCH,** and **positioned UPDATE and DELETE**)

As previously noted, you could beef up the treatment of SELECT here if you like, by bringing in some of the material from Chapter 8 (Section 8.6).

Stress the point that ORDER BY isn't a relational operation (because its result isn't a relation). This fact doesn't mean that ORDER BY isn't useful, but it does mean it isn't part of the relational algebra or calculus (see Chapters 7 and 8), or more generally the relational model.

Examples and certain minor details in this section are based on PL/I ("for definiteness"). As in Chapter 2, you can substitute (e.g.) C for PL/I if you prefer; however, you should be aware that some of the specifics need rather more substantial revision if the host language happens to be Java. Further details are beyond the scope of both the book and this manual.

Here's an oddity you might want to be aware of (though I certainly wouldn't discuss it in class unless anyone raises the issue). Consider:

```
DECLARE CURSOR C1 FOR SELECT S# FROM SP ORDER BY S# ... ;
/* the "..." stands for a FOR UPDATE clause -- excluded */
/* here because it isn't discussed in the chapter       */

OPEN C1 ;
FETCH C1 ... ;
DELETE SP WHERE CURRENT OF C1 ;
```

Which specific SP row is deleted? The standard doesn't say! (Specifically, it doesn't say it's the row the cursor is positioned on. And if you think about it, there's no way it *could* say that, because there's no way to identify which row that is.)

Another point you should be aware of, though again I wouldn't mention it unless asked: SQL tables can have duplicate column names! Here's a trivial illustration:

```
SELECT S.S#, SP.S#
FROM   S, SP
WHERE  ... ;
```

The result of this query has two columns, both of which are called S#. *Note:* Further discussion of this issue and many related ones (and the problems such considerations can lead to) can be found in an article by myself, "A Sweet Disorder," due to be published soon on the website *http://www.dbdebunk.com* (probably before the book itself is published).

4.7 Dynamic SQL and SQL/CLI

The topics of this section can be skipped if desired; the book deliberately doesn't go very deep, anyway (the topics are full of messy details that don't really belong in a textbook like this one).

4.8 SQL Isn't Perfect

The sole paragraph in this section in the book says it all. The message is important, though.

Answers to Exercises

As already mentioned, Fig. 4.5 is repeated (along with Fig. 3.8) inside the back cover of the book, for ease of subsequent reference.

4.1
```
CREATE TYPE S# ... ;
CREATE TYPE P# ... ;
CREATE TYPE J# ... ;
CREATE TYPE NAME ... ;
CREATE TYPE COLOR ... ;
CREATE TYPE WEIGHT ... ;
CREATE TYPE QTY ... ;

CREATE TABLE S
     ( S#      S#,
       SNAME   NAME,
       STATUS  INTEGER,
       CITY    CHAR(15),
     PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
     ( P#      P#,
       PNAME   NAME,
       COLOR   COLOR,
       WEIGHT  WEIGHT,
       CITY    CHAR(15),
```

```
        PRIMARY KEY ( P# ) ) ;

    CREATE TABLE J
         ( J#      J#,
           JNAME   NAME,
           CITY    CHAR(15),
         PRIMARY KEY ( J# ) ) ;

    CREATE TABLE SPJ
         ( S#      S#,
           P#      P#,
           J#      J#,
           QTY     QTY,
         PRIMARY KEY ( S#, P#, J# ),
         FOREIGN KEY ( S# ) REFERENCES S,
         FOREIGN KEY ( P# ) REFERENCES P,
         FOREIGN KEY ( J# ) REFERENCES J ) ;
```

4.2 *No answer provided.*

4.3 *No answer provided.*

4.4 a.
```
INSERT INTO S ( S#, SNAME, CITY )
       VALUES ( S# ('S10'), NAME ('Smith'), 'New York' ) ;
```

STATUS here is set to the applicable default value (see Chapter 6, Section 6.6).

b.
```
DELETE
FROM   J
WHERE  J# NOT IN
     ( SELECT J#
       FROM   SPJ ) ;
```

Note the nested subquery and the **IN** operator (actually, the *negated* IN operator) in this solution. See Section 8.6 for further explanation.

c.
```
UPDATE P
SET    COLOR = 'Orange'
WHERE  COLOR = 'Red' ;
```

4.5 Note first that there might be some suppliers who supply no projects at all; the following solution deals with such suppliers satisfactorily. How, exactly? *Answer:* By printing supplier details followed by *no* project details——i.e., it does at least print the supplier information. *Note to the instructor:* Avoid getting sidetracked into a discussion of outer join here! We'll get to that deprecated operator in Chapter 19. (Note in

particular that it's not a *relational* operator, because it yields a result that's not a relation.)

First we define two cursors, CS and CJ, as follows:

```
EXEC SQL DECLARE CS CURSOR FOR
          SELECT S.S#, S.SNAME, S.STATUS, S.CITY
         FROM   S
         ORDER  BY S# ;

EXEC SQL DECLARE CJ CURSOR FOR
         SELECT J.J#, J.JNAME, J.CITY
         FROM   J
         WHERE  J.J# IN
              ( SELECT SPJ.J#
                FROM   SPJ
                 WHERE  SPJ.S# = :CS_S# )
         ORDER  BY J# ;
```

Note the nested subquery and the IN operator once again.

When cursor CJ is opened, host variable CS_S# will contain a supplier number value, fetched via cursor CS. The procedural logic is essentially as follows (pseudocode):

```
EXEC SQL OPEN CS ;
DO for all S rows accessible via CS ;
   EXEC SQL FETCH CS INTO :CS_S#, :CS_SN, :CS_ST, :CS_SC ;
   print CS_S#, CS_SN, CS_ST, CS_SC ;
   EXEC SQL OPEN CJ ;
   DO for all J rows accessible via CJ ;
      EXEC SQL FETCH CJ INTO :CJ_J#, :CJ_JN, :CJ_JC ;
      print CJ_J#, CJ_JN, CJ_JC ;
   END DO ;
   EXEC SQL CLOSE CJ ;
END DO ;
EXEC SQL CLOSE CS ;
```

4.6 The basic problem here is this: We need to "explode" the given part to *n* levels, but we don't know the value of *n*. Now, SQL:1999 introduced the ability to write *recursive expressions*. Using that feature, we can formulate the query as follows:

```
WITH RECURSIVE TEMP ( MINOR_P# ) AS
 ( ( SELECT MINOR_P#                  /* initial subqueryy  */
     FROM   PART_STRUCTURE
     WHERE  MAJOR_P# = :GIVENP# )
   UNION
   ( SELECT PP.MINOR_P#               /* recursive subquery */
     FROM   PP, TEMP
     WHERE  PP.MAJOR_P# = TEMP.MINOR_P# ) )
```

```
    SELECT DISTINCT MINOR_P#              /* final subquery     */
    FROM   TEMP ;
```

If recursive expressions aren't supported, however, we'll have to write a program to do the job. We might consider a recursive program like the following (pseudocode):

```
    CALL RECURSION ( GIVENP# ) ;

    RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
        DCL UPPER_P# ... ;
        DCL LOWER_P# ... ;
        EXEC SQL DECLARE C "reopenable" CURSOR FOR
                 SELECT MINOR_P#
                 FROM   PART_STRUCTURE
                 WHERE  MAJOR_P# = :UPPER_P# ;

        print UPPER_P# ;
        EXEC SQL OPEN C ;
        DO for all PART_STRUCTURE rows accessible via C ;
           EXEC SQL FETCH C INTO :LOWER_P# ;
           CALL RECURSION ( LOWER_P# ) ;
        END DO ;
        EXEC SQL CLOSE C ;
    END PROC ;
```

Each recursive invocation here creates a new cursor; we've assumed that the (fictitious) specification "reopenable" on DECLARE CURSOR means it's legal to OPEN that cursor even if it's already open, and that the effect of such an OPEN is to create a new ***instance*** of the cursor for the specified table expression (using the current values of any host variables referenced in that expression). We've assumed further that references to such a cursor in FETCH (etc.) are references to the "current" instance, and that CLOSE destroys that instance and reinstates the previous instance as "current." In other words, we've assumed that a reopenable cursor forms a ***stack,*** with OPEN and CLOSE serving as the "push" and "pop" operators for that stack.

Unfortunately, these assumptions are purely hypothetical today. There's no such thing as a reopenable cursor in SQL today (indeed, an attempt to OPEN a cursor that's already open will fail). The foregoing code is illegal. But the example makes it clear that "reopenable cursors" would be a very desirable extension to existing SQL.*

---

* We note in passing that a solution very like the one just shown is possible in SQLJ [4.7]——i.e., if the host language is

Java——because cursors in SQLJ are replaced by Java "iterator objects" that *can* be stacked in recursive calls (thanks to an anonymous reviewer for these observations).

———————

Since the foregoing approach doesn't work, we give a sketch of a possible (but very inefficient) approach that does:

```
CALL RECURSION ( GIVENP# ) ;

RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
    DCL UPPER_P# ... ;
    DCL LOWER_P# ... INITIAL ( '      ' ) ;
    EXEC SQL DECLARE C CURSOR FOR
                      SELECT MINOR_P#
                      FROM   PART_STRUCTURE
                      WHERE  MAJOR_P# = :UPPER_P#
                      AND    MINOR_P# > :LOWER_P#
                      ORDER  BY MINOR_P# ;

    print UPPER_P# ;
    DO "forever" ;
       EXEC SQL OPEN C ;
       EXEC SQL FETCH C INTO :LOWER_P# ;
       EXEC SQL CLOSE C ;
       IF no "lower P#" retrieved THEN RETURN ; END IF ;
       IF "lower P#" retrieved THEN
          CALL RECURSION ( LOWER_P# ) ; END IF ;
    END DO ;
END PROC ;
```

Observe in this solution that the same cursor is used on every invocation of RECURSION. (By contrast, new instances of the variables UPPER_P# and LOWER_P# are created dynamically each time RECURSION is invoked; those instances are destroyed at completion of that invocation.) Because of this fact, we have to use a trick——

```
... AND MINOR_P# > :LOWER_P# ORDER BY MINOR_P#
```

——so that, on each invocation of RECURSION, we ignore all immediate components (LOWER_P#s) of the current UPPER_P# that have already been processed.

Additional notes:

a. Reference [4.4] includes an extensive discussion of an alternative approach to problems like this one, plus a brief description of the (nonrelational) Oracle CONNECT BY and START

WITH extensions, which are also intended to address this kind of problem. (See the short paper "The Importance of Closure" in my book ***Relational Database Writings 1991-1994,*** Addison-Wesley, 1995, for an explanation of why the Oracle extensions are indeed, as just claimed, nonrelational.)

b. Reference [4.8] includes a lengthy discussion of the approach adopted by IBM's DB2 to recursive queries. SQL:1999's recursive expressions are based on the IBM approach. The IBM approach is unfortunately subject to a large number of restrictions that are hard to understand, explain, justify, or remember; fortunately, the SQL:1999 support relaxes most if not all of those IBM restrictions.

c. Chapter 7 (end of Section 7.8) describes a pertinent relational operator called ***transitive closure***.

***** End of Chapter 4 *****

P A R T I I

T H E R E L A T I O N A L M O D E L

The relational model is the foundation of modern database technology; it's what makes the field a science. Thus, any book on the fundamentals of database technology **must** include thorough coverage of the relational model, and any database professional **must** understand the relational model in depth. Of course, the material isn't "difficult," but (to repeat) it ***is*** the foundation, and it will remain so for as far out as anyone can see (claims to the contrary from advocates of object orientation, XML, and other such technologies notwithstanding).

Note carefully, however, that the relational model isn't a static thing——it has evolved and expanded over the years and continues to do so. This part of the book reflects the current thinking of myself and other workers in this field (and the treatment is meant to be fairly complete, even definitive, as of the time of writing), but it should ***not*** be taken as the last word on the subject; further evolutionary developments can certainly be expected. By way of example, see the discussion of temporal data in Chapter 23 of the present book.

The chapters are as follows:

5. Types
6. Relations
7. Relational Algebra
8. Relational Calculus
9. Integrity
10. Views

Throughout these chapters, we use the formal relational terminology of relations, tuples, attributes, etc. (except in the SQL sections, where we naturally use SQL's own terms——tables, rows, columns, etc.).

The chapters are, regrettably, very long (this part of the book is almost a book in its own right); however, the length reflects the importance of the subject matter. ALL CHAPTERS (with the possible exception of Chapter 8) MUST BE COVERED ***CAREFULLY*** AND ***THOROUGHLY:*** Everything else builds on this material, and it mustn't be skipped or skimped or skimmed, except possibly as indicated in the notes on individual chapters. (However, detailed treatment of Chapter 5 might be deferred. See the specific notes on that chapter.)

Copyright (c) 2003 C. J. Date

Note: This part of the book is the part above all others that distinguishes this book from its competitors. While other database books do deal with the relational model (of course!), they mostly seem to treat it as just another aspect of the overall subject of database technology (like, e.g., security, or recovery, or "semantic modeling"), and thus typically fail to emphasize the relational model's crucial role as the **foundation**. They also usually fail to explain the important issue of **interpretation** (the predicate stuff). Sometimes, they even get significant details *wrong* ... No names, no pack drill.

Finally, a word regarding SQL. We've already seen that SQL is the standard "relational" database language, and just about every database product on the market supports it (or, more accurately, some dialect of it [4.22]). As a consequence, no modern database book would be complete without fairly extensive coverage of SQL. The chapters that follow on various aspects of the relational model therefore do also discuss the relevant SQL facilities, where applicable (they build on Chapter 4, which covers basic SQL concepts). Other aspects of SQL are likewise covered in sections in the relevant chapters later in the book.

A couple of further points:

- One reviewer of the previous edition of the book suggested that "as commercial products support more features or aspects of the relational model," I keep "raising the bar," thereby putting "my" relational model always out of reach. "This is fine because it results in better commercial products. However, ... it also makes it difficult for the reader to be sympathetic [to] Date's criticisms of commercial products."

  I'd like to respond to this comment. I don't think I do keep "raising the bar." I certainly do try to keep improving my explanations of what the relational model is, but I don't think those improved explanations reflect substantial changes to the model as such; I would say rather that they merely reflect improvements in my own understanding. What's more, what changes have occurred in those explanations have, I think, always been "backward compatible"; I don't think a commercial product that implemented the model as I first described it would be precluded in any significant way from supporting the model as I see it now.

- Anyone who tries to teach the relational model from this book will almost certainly be familiar already with the notion of *nulls* (in particular, with nulls as supported in SQL). Please be aware, therefore, that **I categorically reject nulls,** for numerous good reasons. Some of those reasons are explained in

detail in Chapter 19; here let me just say that (*pace* Codd) a relation that "contains nulls" isn't a relation, and "the relational model with nulls" isn't the relational model. So (to spell the point out), whenever I use the term "the relational model," I mean quite categorically something that doesn't include any nulls.

In accordance with the foregoing, the definitions and discussions and examples in this part of the book all assume, tacitly, that there's no such thing as a null. There are, inevitably, one or two forward references to Chapter 19, but the point I'm trying to make is that the instructor shouldn't be tempted into falling into either:

a. The trap of thinking that I'd forgotten about nulls

b. The trap of trying to "embellish" the material by adding anything (anything positive, that is!) having to do with nulls

Indeed, it's my very strong opinion that nulls are a mistake and should never have been introduced at all, but it would be wrong in a book of this nature to ignore them entirely; that's why Chapter 19 is included.

***** End of Introduction to Part II *****

Copyright (c) 2003 C. J. Date

Chapter 5

T y p e s

Principal Sections

- Values *vs.* variables
- Types *vs.* representations
- Type definition
- Operators
- Type generators
- SQL facilities

General Remarks

This chapter is new in this edition (it's a greatly expanded and completely rewritten version of portions of Chapter 5 from the seventh edition). It opens with this remark:

> *Note: You might want to give this chapter a "once over lightly" reading on a first pass. The chapter does logically belong here, but large parts of the material aren't really needed very much prior to Chapter 20 in Part V and Chapters 25-27 in Part VI.*

From a teaching point of view, therefore, you might want to just take types as a given for now and go straight on to Chapter 6. If you do, however, you'll need to come back to this material before covering any of Chapters 20 and 25-27, and you'll need to be prepared for occasional questions prior to that point on the topics you've temporarily skipped.

As noted in the introduction in this manual to this part of the book, it's this part above all others that I believe distinguishes this book from the competition. With respect to this chapter specifically, one feature that sets the book apart from others (including previous editions of *this* book) is its emphasis on **domains as types**. The chapter goes into considerable detail on what's involved in defining——and, to some extent, implementing——such types (associated operators included). *The stated position that a domain and a type are the same thing permeates the entire book from this point forward;* in fact, I prefer the term *type,* and use *domain* mostly just in contexts where history demands it.

Copyright (c) 2003 C. J. Date

So we're talking about ***type theory***. Type theory is really a programming language topic; however, it's highly relevant to database theory, too (in fact, it provides a basis on which to build such a theory). It might be characterized as the point where "databases meet programming languages." It seems to me that the database community ignored this stuff for far too long, to their cost (to ours too, as users). I could certainly quote some nonsense from the database literature in this connection. For example:

(Begin quote)

Even bizarre requests can easily be stated; for example,

```
SELECT c.customer_name
FROM   Customer_Table c, Zoo_animal_Table z
WHERE  c.no_of_children = z.no_of_legs
AND    c.eye_color = z.eye_color ;
```

This request joins the Customer_Table and Zoo_animal_Table relations based on relationships phrased in terms of no_of_children, no_of_legs, and eye_color. The meaning of these relationships is not entirely clear.

(End quote)

This quote is taken from a book on object databases; I'll leave it as an exercise for you to deconstruct it.

By way of a second example, I could simply point to the mess the SQL standard has made of this whole issue (see Section 5.7 in this chapter, also the "SQL Facilities" sections in Chapters 6, 9, 19, 20, and 26).

The approach we advocate (to databases overall), then, is founded on four core concepts: ***type, value, variable, operator***. These concepts are NOT novel (I like to say "they're not new and they'll never be old"). Of them, **type** is the most fundamental ... To see why, consider type INTEGER (this example is taken from the annotation to reference [3.3], ***The Third Manifesto***):

- The integer "3" might be a ***value*** of that type.
- N might be a ***variable*** of that type, whose value at any given time is some integer value (i.e., some value of that type).
- And "+" might be an ***operator*** that applies to integer values (i.e., to values of that type).

Basic point: Operands for a given operator must be of the right type (give examples). We assume throughout that type checking is done at compile time, wherever possible. Points arising:

- Types (domains) are ***not*** limited to being scalar types only, though people often think they are.

- We prefer the term ***operator*** over the "equivalent" term ***function***. One reason for that preference is that all functions are operators, but not all operators are functions. For further discussion, see reference [3.3].

- Following on from the previous point: Please note that we're ***not*** following SQL usage here, which makes a purely syntactic distinction between ***operator*** and ***function***. To be specific, SQL uses ***function*** to mean an operator that's invoked by means of classical functional notation——or an approximation to that notation, at any rate!——and ***operator*** to mean one that's invoked using a special prefix or infix notation (as in, e.g., prefix or infix minus). We're not very interested in matters that are primarily syntactic in nature.

Types can be **system-defined** (built in) or **user-defined**.

Note right up front that "=" and ":=" ***must*** be defined for every type. You can use this fact to introduce the important notion of ***overloading*** (meaning different operators with the same name). Note too that ***v1*** = ***v2*** if and only if ***v1*** and ***v2*** are in fact **the very same value** (our "=" is really ***identity***). This point is worth repeating whenever it makes sense to do so, until the point is crystal clear and second nature to everyone. ***Aside:*** SQL allows "=" to have user-defined semantics! ... and possibly not to be defined at all! ... at least for structured types. Note some of the implications: Can't specify "uniqueness" on columns containing such values ... Can't do joins over such columns ... Can't do GROUP BY on such columns ... And so on.

Forward reference to Chapter 6: The relational model doesn't prescribe specific types——with one exception, type BOOLEAN (the most fundamental type of all).* In other words, the question as to what data types are supported is orthogonal to the question of support for the relational model as such. Many people, and products, are confused over this simple point, typically claiming that "the relational model can support only simple types like numbers and strings." Not so! There's a nice informal jingle that can serve to reinforce this message: ***Types are orthogonal to tables*** [3.3].

———————

* Of course, it also prescribes one type *generator,* RELATION, but a type generator isn't a type.

———————

Some text from Chapter 26 to illustrate the foregoing popular misconception:

> The following quotes are quite typical: "Relational database systems support a small, fixed collection of data types (e.g., integers, dates, strings)" [26.34]; "a relational DBMS can support only ... its built-in types [basically just numbers, strings, dates, and times]" [25.31]; "object/relational data models [*sic*] extend the relational data model by providing a richer type system" [16.21]; and so on.

(To explain that "[*sic*]": We'll argue in Chapter 26 that there's only one "object/relational model," and that model is in fact the *relational* model——nothing more and nothing less.)

5.2 Values *vs.* Variables

One of the great **logical differences** (see the preface). We'll be appealing to this particular distinction (between values and variables) many, many times in the chapters to come. It's IMPORTANT, and a great aid to clear thinking.

Every value is of just one type (this becomes "just one *most specific* type" when we get to inheritance in Chapter 20; declared types are always known at compile time, but "most specific types" might not be). Every variable is of just one type, too, called the *declared* type. Relational attributes, read-only operators, parameters, and expressions in general also all have a declared type. Give examples.

5.3 Types *vs.* Representations

Another of the great logical differences!——and one that SQL in particular gets confused over. (Actually SQL gets confused over values *vs.* variables as well.)

Carefully explain:

- **Scalar** *vs.* **nonscalar** types (and values and variables etc.)
- **Possible representations** ("possreps")

- **Selectors**

- **THE_ operators**

Regarding ***scalar types:*** A scalar type is one that has no user-visible components. Note carefully, however, that fairly complicated things can be scalar values! Chapter 5 includes examples in which (e.g.) geometric points and line segments are legitimately regarded as scalar values. Make sure the students understand the difference between components of a type ***per se*** and components of a ***possible representation*** for a type. Sometimes we do sloppily say things like "the X component of point P," but what we should really be saying is "the X component of ***a certain possible representation of*** point P."

Regarding ***possreps:*** The distinction between ***physical*** (or ***actual***) and ***possible*** representations is much blurred in the industry (especially in SQL), but it really ought not to be. It's crucial to, and permeates, the proposals of reference [3.3].

Regarding ***selectors:*** Selectors are a generalization of ***literals***. And, of course, "everyone knows" what a literal is——or do they? Certainly it seems to be hard to find a definition of the concept in the literature (good or bad); in fact, there seems to be some ***confusion*** out there (see ODMG, for example). Here's a ***good*** definition (from ***The Third Manifesto***):

> A literal is a symbol that denotes a value that's fixed and determined by the particular symbol in question (and the type of that value is also fixed and determined by the symbol in question). Loosely, we can say that a literal is ***self-defining***.

Note: The term ***selector*** is nonstandard——it comes from reference [3.3], of course——but there doesn't seem to ***be*** a standard term for the concept. Don't confuse it with a ***constructor;*** a constructor, at least as that term is usually understood, constructs a ***variable,*** but a selector selects a ***value*** (SQL has its own quirks in this area, however, as we'll see).

Regarding ***THE_ operators:*** Once again these ideas come from reference [3.3]. Note, however, that most commercial products support, not THE_ operators as such, but rather "GET_ and SET_ operators" of some kind (possibly, as in SQL, via dot qualification syntax). The distinction is explained in detail in reference [3.3]; in a nutshell, however, GET_ and THE_ ***operators*** are the same thing (just different spellings), but SET_ operators and THE_ ***pseudovariables*** are **not** the same thing (because SET_ operators are typically defined to have a return value). Note

that, by definition, THE_ *operator* invocations appear in source positions——typically the right side of an assignment——while THE_ *pseudovariable* invocations appear in target positions——typically the left side of an assignment.

Further explanation: Essentially, a pseudovariable is an operational expression (not just a simple variable reference) that can appear in a target position. The term is taken from PL/I. SUBSTR provides a PL/I example.

Note: This section of the book includes the following: "Alternatively, THE_R and THE_θ could be defined directly in terms of the protected operators (details left as an exercise)." Here's an answer to that exercise:

```
OPERATOR THE_R ( P POINT ) RETURNS ( RATIONAL ) ;
   BEGIN ;
      VAR X RATIONAL ; VAR Y RATIONAL ;
      X  :=  X component of physical representation of P ;
      Y  :=  Y component of physical representation of P ;
      RETURN ( SQRT ( X ** 2 + Y ** 2 ) ) ;
   END ;
END OPERATOR ;

OPERATOR THE_θ ( P POINT ) RETURNS ( RATIONAL ) ;
   BEGIN ;
      VAR X RATIONAL ; VAR Y RATIONAL ;
      X  :=  X component of physical representation of P ;
      Y  :=  Y component of physical representation of P ;
      RETURN ( ARCTAN ( Y / X ) ) ;
   END ;
END OPERATOR ;
```

5.4 Type Definition

Distinguish types introduced via *the TYPE statement* and types obtained by invoking some *type generator*. They're all types, of course, and can all be used wherever a type is needed; however, I note in passing that an analogous remark does *not* apply to SQL (were you surprised?). This section is concerned with the former case only, and with scalar types only.

Explain **type constraints** carefully (forward reference to Chapter 9). Obviously fundamental——but SQL doesn't support them! (Forward reference to Section 5.7.)

5.5 Operators

Carefully explain:

- **Read-only** *vs.* **update** operators
- **THE_ pseudovariables** (if not already covered)
- **Multiple assignment**
- **Strong typing** (if not already covered)

Read-only *vs.* Update Operators

The distinction between read-only and update operators——another logical difference!——becomes particularly important when we get to *inheritance* (Chapter 20). Prior to that point, it's mostly common sense. You should be aware, however, that the idea that update operators return no value and must be invoked by explicit CALLs isn't universally accepted but is required——for good reasons——by the type model adopted in *The Third Manifesto* [3.3].

A note regarding the REFLECT example: The alert student might notice that REFLECT is in fact a *scalar* update operator specifically, and might object, correctly, that scalar update operators aren't part of the relational model. The discussion of such operators in this chapter might thus be thought a little out of place. The point is, however, that such operators are definitely needed as part of the total environment that surrounds any actual implementation of the relational model.* (Also, there really isn't any other sensible place in the book to move the discussion to!)

———————

* In the same kind of way, *scalar variables* aren't part of the relational model but will surely be available in any environment in which a relational implementation exists. For example, a scalar variable will be needed to serve as a receiver for any scalar value that might be retrieved from some tuple in some relation in the database.

———————

By the way: If a given type has no operators other than the prescribed ones ("=", ":=", selectors, THE_ operators, plus a few more to be defined in subsequent chapters), then the type probably wasn't worth defining in the first place.

Multiple Assignment

Assignment as such is the only update operator logically required (all other update operators are just shorthand for certain assignments, as we already know in the case of relational assignments specifically). ***Multiple*** assignment is somewhat novel and not fully supported in today's products, but we believe it's logically required. Basic idea is to allow several individual assignments to be executed (a) without any integrity checking being done until the end and at the same time (b) without the application being able to see any temporarily inconsistent state of the database "in the middle" of those individual assignments. A couple of points arising:

- As the book says, the semantics are carefully specified to give a well-defined result when distinct individual assignments update distinct parts of the same target variable (an important special case). It's ***not*** worth getting into this issue in a live presentation, but here are the rules for purposes of reference:

  Let *MA* be the multiple assignment

  A1 , A2 , ... , An ;

  Then the semantics of *MA* are defined by the following four steps (pseudocode):

  Step 1: For *i* := 1 to *n,* we can consider *Ai* to take the form (after syntactic substitution, if necessary)

  Vi := *Xi*

  where *Vi* is the name of some declared variable and *Xi* is an expression whose declared type is some subtype of that of *Vi.*

  Step 2: Let *Ap* and *Aq* (*p* < *q*) be such that (a) *Vp* and *Vq* are identical and (b) there is no *Ar* (*r* < *p* or *p* < *r* < *q*) such that *Vp* and *Vr* are identical. Replace *Aq* in *MA* by an assignment of the form

  Vq := WITH *Xp* AS *Vq* : *Xq*

  and remove *Ap* from *MA.* Repeat this process until no such pair *Ap* and *Aq* remains. Let *MA* now consist of the sequence

  U1 := *Y1* , *U2* := *Y2* , ... , *Um* := *Ym* ;

  where each *Ui* is some *Vj* (1 ≤ *i* ≤ *j* ≤ *m* ≤ *n*).

Step 3: For *i* := 1 to *m*, evaluate *Yi*. Let the result be *yi*.

Step 4: For *i* := 1 to *m*, assign *yi* to *Ui*. ▉

- As already indicated, multiple assignment is unorthodox but important. In fact, it will become "more orthodox" with SQL:2003, which will explicitly introduce such a thing (albeit not for ***relational*** assignment, since SQL doesn't support relational assignment at all as yet):

  SET (*target list*) = *row* ;

  The odd thing is that SQL in fact does already support multiple assignment explicitly in the case of the UPDATE statement. What's more, it supports multiple ***relational*** assignment as well, implicitly, in at least two situations:

  1. As part of its support for referential actions such as ON DELETE CASCADE

  2. As part of its (limited) support for updating (e.g.) join views

Values can be converted from one type to another by means of explicit ***CAST*** operators or by ***coercion*** (implicit, by definition; the point is worth making that coercions are——presumably——possible only where explicit CASTs are possible). The book adopts the conservative position that coercions are illegal, however (for reasons of both simplicity and safety); thus, it requires comparands for "=" to be of the same type, and it requires the source and target in ":=" to be of the same type (in both cases, until we get to Chapter 20).

Forward reference to Chapter 26: Domains, types, and **object classes** are all the same thing; hence, domains are the key to marrying object and relational technologies (i.e., the key to "object/relational" database systems).

Draw the attention of students to the use of WITH (in the definition of the DIST operator), if you haven't already done so. We'll be using this construct a lot.

5.6 Type Generators

Type generators and corresponding generated types are known by many different names. Type generators have generic "possreps" and generic operators and generic constraints. Illustrate these ideas with reference to ARRAY (as in the book) or your own preferred

type generator ... You should be aware that we will be introducing an important type generator, INTERVAL, in Chapter 23.

Most generated types are nonscalar (but not all; SQL's REF types are a counterexample, as are the interval types discussed in Chapter 23).

Two (definitely nonscalar) type generators of particular importance in the relational world are TUPLE and RELATION, to be discussed in the next chapter.

5.7 SQL Facilities

Quite frankly, this section is much longer than it ought to be, thanks to SQL's extreme lack of orthogonality (numerous special cases, constructs with overlapping but not identical functionality, unjustified exceptions, etc., all of which have to be individually explained). In fact, this very state of affairs could be used to introduce and justify the concept of orthogonality (the book itself doesn't explain this concept until Chapter 8, Section 8.6, though the term is mentioned in passing in Chapter 6).

Regarding ***SQL built-in types:*** Note that bit string types were added in SQL:1992 and will be dropped again in SQL:2003! There are other examples of this phenomenon, too. Why? (Rhetorical question ... Could the answer have anything to do with "design by committee"?) You might want to note too that almost nobody——actually nobody at all, so far as I know——has implemented type BOOLEAN (which I earlier called "the most fundamental type of all"). As a consequence, certain SQL expressions——in particular, those in WHERE and HAVING clauses——return a value of a type that's unknown in the language (!).

The fact that SQL already supports a limited form of multiple assignment is worth noting, if you haven't already mentioned it.

Regarding ***DISTINCT types:*** Note all of the violations of orthogonality this construct involves! Might be an interesting exercise to list them.

Regarding ***structured types:*** Now this is a big topic. This chapter covers the basics, but we'll have a lot more to say in Chapter 6 (where we discuss the idea of defining tables to be "of" some structured type); Chapter 20 (where we discuss SQL's approach to type inheritance, which applies to structured types only); and Chapter 26 (where we discuss the use of structured types in SQL's approach to "object/relational" support).

Here's an example to explain——or at least illustrate——SQL's "mutators" (mention "observers" too). Consider the following assignment statement:

```
SET P.X = Z ;
```

P here is of type POINT and has "attributes" [*sic*] X and Y. This assignment is defined to be equivalent to the following one:

```
SET P = P.X ( Z ) ;
```

The expression on the right side here invokes the "mutator" X on the variable P and passes it as argument Z. That mutator invocation returns a point value identical to that previously contained in P, except that the X attribute is whatever the value of Z is. That returned point value is then assigned back to P.

Note, therefore, that SQL's "mutators" don't really do any "mutating"!——they're really *read-only* operators, and they return a value. But that value can then be assigned to the relevant variable, thereby achieving the conventional "mutation" effect as usually understood. As an exercise, you might want to think about the implications of this approach for the more complicated assignment

```
SET LS.BEGIN.X = Z ;
```

(Try writing out the expanded form for which this is just a shorthand.)

In a somewhat similar manner, SQL's "constructors" aren't exactly constructors as usually understood in the object world. In particular, they return values, not variables (and they don't allocate any storage).

It is very unclear as to (a) why SQL allows some types not to have an "=" operator and (b) why it allows the semantics of that operator to be user-defined when it does exist.

Regarding *type generators* (ROW and ARRAY): There are many oddities here, some of which are noted in the text.

Here are some further weirdnesses of which, as an instructor, you probably ought at least to be aware:

1. *Assignment doesn't always mean assignment:* Suppose X3 is of type CHAR(3) and we assign the string 'AB' to it. After that assignment, the value of X3 is actually 'AB ' (note the trailing blank), and the comparison X3 = 'AB' won't give TRUE if NO PAD is in effect (see reference [4.20]). *Note:* Many

similar but worse situations arise when nulls are taken into account (see Chapter 19).

2. ***Equality isn't always equality:*** Again suppose X3 is of type CHAR(3) and we assign the string 'AB' to it. After that assignment, the value of X3 is actually 'AB ' (note the trailing blank), but the comparison X3 = 'AB' ***does*** give TRUE if PAD SPACE is in effect (again, see reference [4.20]).* ***Note:*** There are many other situations in SQL in which two values *x* and *y* are distinct and yet the comparison *x* = *y* gives TRUE. This state of affairs causes great complexity in, e.g., uniqueness checks, GROUP BY operations, DISTINCT operations, etc.

---

* At the same time X3 LIKE 'AB' gives FALSE ... so two values can be equal but not "like" each other! Some people find this state of affairs amusing (Lewis Carroll, perhaps?).

---

3. The following text is taken from Section 5.7:

   (Begin quote)

   [We] could define a function——a polymorphic function, in fact——called ADDWT ("add weight") that would allow two values to be added regardless of whether they were WEIGHT values or DECIMAL(5,1) values or a mixture of the two. All of the following expressions would then be legal:

```
ADDWT ( WT, 14.7 )
ADDWT ( 14.7, WT )
ADDWT ( WT, WT )
ADDWT ( 14.7, 3.0 )
```

   (End quote)

   Note, however, that——even if the current value of WT is WEIGHT(3.0)——the four invocations aren't constrained to return the same value, or even values of the same type, or even of compatible types! The reason is that we're talking here about ***overloading*** polymorphism, not ***inclusion*** ditto (see Chapter 20); the four ADDWTs are thus really four different functions, and their semantics are up to the definer in each case.

Copyright (c) 2003 C. J. Date

4. Suppose X and Y are of type POINT and "=" hasn't been defined for that type. In order to determine whether X and Y are in fact equal, we can see whether the expression

```
X.X = Y.X AND X.Y = Y.Y
```

gives TRUE. Likewise, if X and Y are of type LINESEG and "=" hasn't been defined for either POINT or LINESEG, we can see whether the following expression

```
X.BEGIN.X = Y.BEGIN.X AND
X.BEGIN.Y = Y.BEGIN.Y AND
X.END.X   = Y.END.X   AND
X.END.Y   = Y.END.Y
```

gives TRUE (and so on). *Your comments here.*

Answers to Exercises

5.1 For assignment, the declared types of the target variable and the source expression must be the same. For equality comparison, the declared types of the comparands must be the same. *Note:* Both of these rules will be refined somewhat in Chapter 20.

5.2 For value *vs.* variable, see Section 5.2. For type *vs.* representation, see Section 5.3. For physical *vs.* possible representation, see Section 5.3, subsection "Possible Representations, Selectors, and THE_ Operators." For scalar *vs.* nonscalar, see Section 5.3, subsection "Scalar *vs.* Nonscalar Types"; see also elaboration below. For read-only *vs.* update operators, see Section 5.5.

Note: As a matter of fact, the precise nature of the scalar *vs.* nonscalar distinction is open to some debate. We appeal to that distinction in this book——quite frequently, in fact——because it does seem intuitively useful; however, it's possible that it'll be found, eventually, not to stand up to close scrutiny. The issue isn't quite as clearcut as it might seem.

5.3 Brief definitions:

- **Coercion** is implicit conversion.
- A **generated type** is a type obtained by invocation of a type generator.
- A **literal** is a symbol that denotes a value that's fixed and determined by the particular symbol in question (and the type of that value is also fixed and determined by the symbol in

question). For example, 5 is a literal denoting the fixed value *five* (of type INTEGER); 'literal' is a literal denoting the fixed value *literal* (of type CHAR); WEIGHT (17.0) is a literal denoting the fixed value *weight 17.0* (of type WEIGHT); and so on.

- An **ordinal type** is a type *T* such that the expression *v1* > *v2* is defined for all pairs of values *v1* and *v2* of type *T*.

- An operator is said to be **polymorphic** if it's defined in terms of some parameter *P* and the arguments corresponding to *P* can be of different types on different invocations.

- A **pseudovariable** is an operator invocation appearing in a target position (in particular, on the left side of an assignment). THE_ pseudovariables are an important special case.

- A **selector** is an operator that allows the user to specify or *select* a value of the type in question by supplying a value for each component of some possible representation of that type. Literals are an important special case.

- **Strong typing** means that whenever an operator is invoked, the system checks that the operands are of the right types for that operator.

- A **THE_ operator** provides access to a specified component of a specified possible representation of a specified value.

- A **type generator** is an operator that returns a type.

5.4 Because they're just shorthand——any assignment that involves a pseudovariable is logically equivalent to one that does not.

5.5
```
OPERATOR CUBE ( N RATIONAL ) RETURNS RATIONAL ;
   RETURN ( N * N * N ) ;
END OPERATOR ;
```

5.6
```
OPERATOR FG ( P POINT ) RETURNS POINT ;
   RETURN ( CARTESIAN ( F ( THE_X ( P ) ),
                        G ( THE_Y ( P ) ) ) ) ;
END OPERATOR ;
```

5.7
```
OPERATOR FG ( P POINT ) UPDATES P ;
   THE_X ( P ) := F ( THE_X ( P ) ) ,
   THE_Y ( P ) := G ( THE_Y ( P ) ) ;
END OPERATOR ;
```

Copyright (c) 2003 C. J. Date

Note the multiple assignment here. Note too that there's no explicit RETURN statement; rather, an implicit RETURN is executed when the END OPERATOR statement is reached.

5.8
```
TYPE LENGTH POSSREP { RATIONAL } ;
TYPE POINT  POSSREP { X RATIONAL, Y RATIONAL } ;
TYPE CIRCLE POSSREP { R LENGTH, CTR POINT } ;
   /* R represents (the length of) the radius of the circle */
   /* and CTR the center                                     */
```

The sole selector that applies to type CIRCLE is as follows:

```
CIRCLE ( r, ctr )
/* returns the circle with radius r and center ctr */
```

The THE_ operators are:

```
THE_R ( c )
/* returns the length of the radius of circle c */

THE_CTR ( c )
/* returns the point that is the center of circle c */
```

a.
```
OPERATOR DIAMETER ( C CIRCLE ) RETURNS LENGTH ;
   RETURN ( 2 * THE_R ( C ) ) ;
END OPERATOR ;

OPERATOR CIRCUMFERENCE ( C CIRCLE ) RETURNS LENGTH ;
   RETURN ( 3.14159 * DIAMETER ( C ) ) ;
END OPERATOR ;

OPERATOR AREA ( C CIRCLE ) RETURNS AREA ;
   RETURN ( 3.14159 * ( THE_R ( C ) ** 2 ) ) ;
END OPERATOR ;
```

We're assuming in these operator definitions that (a) multiplying a length by an integer or a rational returns a length* and (b) multiplying a length by a length returns an area (where AREA is another user-defined type).

---

* ***Point for discussion:*** What if the integer or rational is negative or zero?

---

b.
```
OPERATOR DOUBLE_R ( C CIRCLE ) UPDATES C ;
   THE_R ( C )  :=  2 * THE_R ( C ) ;
```

Copyright (c) 2003 C. J. Date

```
    END OPERATOR ;
```

5.9 A triangle can possibly be represented by (a) its three vertices or (b) the midpoints of its three sides.* A line segment can possibly be represented by (a) its begin and end points or (b) its midpoint, length, and slope.

---

* As a subsidiary exercise, you might like to try proving that a triangle is indeed uniquely determined by the midpoints of its sides.

5.10 *No answer provided.*

5.11 *No answer provided.*

5.12 Just to remind you of the possibility, we show one type definition with a nontrivial type constraint:

```
    TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                          CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

(See also Chapter 9.) For simplicity, however, we exclude such CONSTRAINT specifications from the remaining type definitions:

```
    TYPE S#    POSSREP { CHAR } ;
    TYPE P#    POSSREP { CHAR } ;
    TYPE J#    POSSREP { CHAR } ;
    TYPE NAME  POSSREP { CHAR } ;
    TYPE COLOR POSSREP { CHAR } ;
    TYPE QTY   POSSREP { INTEGER } ;
```

We've also omitted the possrep names and possrep component names that the foregoing type definitions would probably require in practice.

5.13 We show a typical value for each attribute. First, relvar S:

```
    S#      :  S# ('S1')
    SNAME   :  NAME ('Smith')
    STATUS  :  20
    CITY    :  'London'
```

Relvar P:

Copyright (c) 2003 C. J. Date

```
    P#      :  P# ('P1')
    PNAME   :  NAME ('Nut')
    COLOR   :  COLOR ('Red')
    WEIGHT  :  WEIGHT (12.0)
    CITY    :  'London'

Relvar J:

    J#      :  J# ('J1')
    JNAME   :  NAME ('Sorter')
    CITY    :  'Paris'

Relvar SPJ:

    S#      :  S# ('S1')
    P#      :  P# ('P1')
    J#      :  J# ('J1')
    QTY     :  QTY (200)
```

5.14

a. Legal; BOOLEAN.

b. Illegal; NAME (THE_NAME (JNAME) || THE_NAME (PNAME)).
   Note: The idea here is to concatenate the (possible) ***character-string representations*** and then "convert" the result of that concatenation back to type NAME. Of course, that conversion itself will fail on a type error, if the result of the concatenation can't be converted to a legal name.

c. Legal; QTY.

d. Illegal; QTY + QTY (100).

e. Legal; INTEGER.

f. Legal; BOOLEAN.

g. Illegal; THE_COLOR (COLOR) = P.CITY.

h. Legal.

5.15 The following observations are pertinent. First, as pointed out at the end of Section 5.4, the operation of defining a type doesn't actually create the corresponding set of values; conceptually, those values already exist, and always will exist (think of type INTEGER, for example). Thus, all the "define type" operation——e.g., the TYPE statement, in **Tutorial D**——really does is introduce a ***name*** by which that set of values can be referenced. Likewise, the DROP TYPE statement doesn't actually drop the

corresponding values, it merely drops the name that was introduced by the corresponding TYPE statement. It follows that "updating an existing type" really means dropping the existing type *name* and then redefining that same name to refer to a different set of values. Of course, there's nothing to preclude the use of some kind of "alter type" shorthand to simplify such an operation (as SQL does, in fact, at least for "structured types").

5.16 Here first are SQL type definitions for the scalar types involved in the suppliers-and-parts database:

```
CREATE TYPE S#     AS CHAR(5)      FINAL ;
CREATE TYPE P#     AS CHAR(6)      FINAL ;
CREATE TYPE J#     AS CHAR(4)      FINAL ;
CREATE TYPE NAME   AS CHAR(20)     FINAL ;
CREATE TYPE COLOR  AS CHAR(12)     FINAL ;
CREATE TYPE WEIGHT AS DECIMAL(5,1) FINAL ;
CREATE TYPE QTY    AS INTEGER      FINAL ;
```

With respect to the question of representing weights in either pounds or grams, the best we can do is define two distinct types with appropriate CASTs (definitions not shown) for converting between them:

```
CREATE TYPE WEIGHT_IN_LBS AS DECIMAL (5,1) FINAL ;

CREATE TYPE WEIGHT_IN_GMS AS DECIMAL (7,1) FINAL ;
```

Types POINT and LINESEG will become "structured" types:

```
CREATE TYPE CARTESIAN AS ( X FLOAT, Y FLOAT ) NOT FINAL ;

CREATE TYPE POLAR AS ( R FLOAT, THETA FLOAT ) NOT FINAL ;

CREATE TYPE LINESEG
         AS ( BB CARTESIAN, EE CARTESIAN ) NOT FINAL ;
```

5.17 See Answer 4.1 and Answer 5.16.

5.18 *No answer provided.*

5.19 See Section 5.7, subsection "Structured Types."

5.20 *No answer provided.*

5.21 Such a type——we call it type *omega*——turns out to be critically important in connection with the type inheritance model defined in reference [3.3]. The details are unfortunately beyond the scope of the book and these answers; see reference [3.3] for further discussion.

Copyright (c) 2003 C. J. Date

5.22 We "explain" the observation by appealing to the SQL standard itself (reference [4.23]), which simply doesn't define any such constructs. As for "justifying" it: We can see no good justification at all. Another consequence of "design by committee"?

5.23 To paraphrase from the body of the chapter: The type designer can effectively conceal the change by a judicious choice of operators. The details are beyond the scope of the book and these answers; suffice it to say that (in my own not unbiased opinion) they're far from being as straightforward as their **Tutorial D** counterparts, either theoretically or in their pragmatic implications.

5.24 There seems to be little logical difference. It isn't clear why CARDINALITY wasn't called COUNT.

***** End of Chapter 5 *****

Copyright (c) 2003 C. J. Date

Chapter 6

R e l a t i o n s

Principal Sections

- Tuples
- Relation types
- Relation values
- Relation variables
- SQL facilities

General Remarks

This chapter is a greatly expanded and completely rewritten version of portions of Chapter 5 from the seventh edition. It contains a very careful presentation of relation types, relation values (relations), and relation variables (relvars). Since relations are made out of tuples, it covers tuple types and values and variables as well, but note immediately that this latter topic isn't all that important in it itself——it's included only because it's needed as a stepping-stone to the former topic. *Caveat:* Relation *values* are made out of tuple *values,* but do NOT make the mistake of thinking that relation *variables* are made out of tuple *variables!* In fact, there aren't any tuple variables in the relational model at all (just as there aren't any scalar variables in the relational model either, as we saw in the previous chapter in this manual).

It's worth saying right up front that relations have attributes, and attributes have types, and the types in question can be **any types whatsoever** (possibly even relation types):

> **The question of what data types are supported is orthogonal to the question of support for the relational model**

Or, more catchily: "Types are orthogonal to tables" (though I'm going to argue later that it would be much better always to talk in terms of *relations,* not tables).

6.2 Tuples

Copyright (c) 2003 C. J. Date

Suggestion: Show the following picture of a tuple as an example and annotate it dynamically to illustrate the formal terms **tuple value** (tuple for short), **component**, **attribute**, **attribute name**, **attribute value**, **attribute type** (and attribute type name), **degree**, **heading**, **tuple type** (and tuple type name). Note in particular that we define an **attribute** to consist specifically of an *attribute-name / type-name pair*. Note further that it will follow (when we get to relvars) that, e.g., attributes S# in relvar S and S# in relvar SPJ are *the same attribute*. This will become important when we get to the relational algebra in Chapter 7.

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P2	P4	7

Note: Attribute values should really be P#('P2'), P#('P4'), QTY(7)——explain. In a similar vein, we often omit the type names when we give *informal* examples of tuple (and relation) headings.

Don't bother to talk through the precise formal definition of "tuple"——just say it's in the book (you can *show* it, if you like, but the point is that, as so often, precise definitions make simple concepts look very complicated).

Show a **Tutorial D** tuple selector invocation, and explain the following important properties of tuples:

- Every tuple contains exactly one value (of the appropriate type) for each of its attributes. *Note:* As an aside, you might want to point out that null is not a value, so right here we have an overwhelming argument against nulls (as usually understood).

- There's no left-to-right ordering to the components.

- Every subset of a tuple is a tuple, and every subset of a heading is a heading——and these remarks are true of the empty subset in particular.

Explain the TUPLE type generator. Probably *don't* get into the point that tuple types have no name apart from the one we already know about of the form TUPLE { *<attribute commalist>* }.

Explain **tuple equality** very carefully (so much depends on it!). As the book says, all of the following are defined in terms of tuple equality:

- Candidate keys (see Chapter 9)
- Foreign keys (see Chapter 9 again)
- Essentially all of the operators of the relational algebra (see Chapter 7)
- Functional and other dependencies (see Chapters 11-13)

and more besides.

The "<" and ">" operators do *not* apply to tuples (explain why——fundamentally because tuples are sets).

Mention tuple projection.

Don't discuss tuple types *vs.* possreps unless someone asks about it ... Even then, I'd probably deal with the issue offline.

6.3 Relation Types

Suggestion: Show the following picture of a relation as an example and annotate it dynamically to illustrate the formal terms **relation value** (relation for short), **attribute**, **attribute name**, **attribute value**, **attribute type** (and attribute type name), **degree**, **cardinality**, **heading**, **body**, **relation type** (and relation type name).

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P1	P2	5
P1	P3	3
P2	P3	2
P2	P4	7
P3	P5	4
P4	P6	8

Note: Attribute values should really be P#('P1') etc. Note that the tuple we talked about in the previous section is a tuple in (the body of) this relation.

Don't bother to talk through the precise formal definition of "relation"——just say it's in the book.

Show a **Tutorial D** relation selector invocation. Every subset of a heading is a heading (as with tuples); every subset of a body is a body. In both cases, the subset in question might be empty.

Explain the RELATION type generator and relation equality.

6.4 Relation Values

This section is perhaps the core of this chapter. State "the four properties" of relations:

1. Relations are normalized.

2. Attributes are unordered, left to right.

3. Tuples are unordered, top to bottom.

4. There are no duplicate tuples.

Now justify them:

1. Regarding ***normalization:*** You should be aware that the history here is somewhat confused (in particular, the first few editions of this book were confused). The true state of affairs is as follows:

   Attribute values are single values, but those values can be absolutely anything.

   We reject the old notion of "value atomicity," on the grounds that it has no absolute meaning——it simply depends on your point of view.* (Draw a parallel with atoms in physics, if you like, which are regarded as indivisible for some purposes but not for others.) Thus, **all** relations are normalized in the relational model——even relations that contain other relations nested inside themselves. It's true that relations with others nested inside themselves are often ***contraindicated,*** but that's a separate point (which we'll be addressing in Chapter 12).

---

* In other words, the concept of "nonatomic values" has never been very clearly defined (certainly it's not very precise). After all, even a number might be decomposed (e.g., into decimal digits, or into integer and fractional parts) in suitable circumstances; so is a number atomic? What about bit and character strings, which are ***obviously*** decomposable? What about dates and times? And so on.

You might want to add that one reason relation-valued attributes (RVAs) are often——though not *always*——contraindicated is that relations involving RVAs are usually asymmetric, leading to complications over query formulation (see Section 11.6 for further discussion). Another is that the predicate for such a relation is often fairly complicated.* For example, consider the relation of Fig. 6.2. That relation shows among other things that supplier S1 supplies the set of parts {P1,P2,P3,P4,P5,P6}. It is thus a "true fact" that supplier S1 supplies the set of parts {P1,P2,P3,P4,P5} ... and the set of parts {P1,P2,P3,P4} ... and the set of parts {P1,P2,P3} ... and many other sets of parts as well (actually 60 others). Doesn't the Closed World Assumption thus require the relation to include tuples corresponding to these additional "true facts" as well? Well, obviously not ... but why not, exactly?

———————

* Some might say it's *second-order*.

———————

Note: Further discussion of the whole issue of all relations being in first normal form (also of relation-valued attributes) can be found in an article by myself, "What Does First Normal Form Really Mean?" (in two parts), to appear soon on the website *http://www.dbdebunk.com* (probably before the book itself is published). Among other things, this article offers some thoughts on the current flurry of interest in the so-called "multi-value" (or "multi-value column") systems, which you might find you need to be aware of in order to fend off certain possible criticisms.

2. Regarding *no attribute ordering:* The book doesn't explicitly make this point, but a good pragmatic argument to justify this property is that, without it, *A* JOIN *B* is different from *B* JOIN *A!* Another is that, in SQL, programs that use "SELECT *" are fragile (they can break in the face of left-to-right column rearrangements in the database——lack of data independence!). *Note:* Further discussion of this issue can be found in another article by myself, "A Sweet Disorder," also due to appear soon on the website *www.dbdebunk.com*.

3. Regarding *no tuple ordering:* The argument that "*n* ways to represent information means *n* sets of operators" (and *n* = 1 is sufficient) is a very strong one. Of course, "no tuple ordering" doesn't mean we can't do ORDER BY ... but it does

mean the *result* of ORDER BY isn't a relation (important point!).

4. Regarding *no duplicate tuples:*

   - A strong *logical* argument here is the one that relies on the fact that tuples are supposed to represent true propositions. If I tell you "The sun is shining outside" and "The sun is shining outside," then I'm simply telling you "The sun is shining outside." If something is true, saying it twice doesn't make it *more* true!

   - One *philosophical* argument is: If things are distinct, they must have distinct identities (quote *The Principle of Identity of Indiscernibles?*[*]); the relational model says let's represent those identities in the same way as everything else (namely, as attribute values within tuples), and then all kinds of good things will happen.

---

[*] If there's no discernible difference between two entities, then there aren't two but only one.

   - One *technical* argument is: Duplicates inhibit the optimizer (because they make expression transformation——*aka* "query rewrite"——harder to do and less widely applicable), thereby leading to worse performance among other things. We'll elaborate on this argument in Chapter 7.

   - Another (and this one is, specifically, an *SQL* argument): Suppose rows *r1* and *r2* are duplicates. If we position a cursor on *r1* (say) and issue a DELETE via that cursor, there's no guarantee——at least according to my reading of the standard——that the effect won't be to delete *r2* instead (!).

Relations *vs.* Tables

The book summarizes some of the main differences between relations and tables. It's worth spending a few minutes on that topic here; in fact, all of the points made in this subsection are worth an airing in a live class. Note that (as the book says) the list of differences is *not* exhaustive; others include (a) the fact that tables are usually thought of as having at least one column (we'll talk about this one in a few minutes); (b) the fact that tables

(at least in SQL) are allowed to include nulls (forward reference to Chapter 19); and (c) the ***horrible*** but widespread perception that "relations are flat" (forward reference to Chapter 22).

Note: The book also makes the point that columns (as opposed to attributes) might have duplicate names, or even no names at all, and asks the question: What are the column names in the result of the following SQL query?

```
SELECT S.CITY, S.STATUS * 2, P.CITY
FROM   S, P ;
```

Answer: Column 1 is called CITY; column 2 has no name; column 3 is called CITY again. ***Note for barrack-room lawyers:*** Actually, the SQL standard does say the implementation is required to assign names to otherwise anonymous columns, but those names are implementation-dependent (they vary from system to system, possibly even from release to release or even more frequently). In any case, those names are also invisible (they're not exposed to the user). Besides, this implementation requirement, even if you believe in it, still doesn't address the problem of ***duplicate*** column names.

Relation-Valued Attributes

Some of the points made in this subsection are probably best made under the earlier discussion of normalization——it probably isn't worth making a separate topic out of them in a live class.

Relations with No Attributes

A gentle introduction to this concept is DEFINITELY worth including as a separate topic. Strong logical justification: TABLE_DEE plays a role in the relational algebra analogous to that played by zero in ordinary arithmetic. ***Don't*** get into details——I think the point's intuitively clear. Can you imagine an arithmetic without zero? Of course not.* Well ... just as you can't imagine an arithmetic without zero, so you shouldn't be able to imagine a relational algebra without TABLE_DEE.

---

* Of course, we did have an arithmetic without zero for many centuries (think of the ancient Romans), but it didn't work very well. In fact, the invention (or discovery) of the concept of zero is arguably one of the great intellectual achievements of the human race.

Operators on Relations

Definitely discuss relation comparisons (including "=" in particular, though it was mentioned previously in Section 6.3). Relation comparisons are another (important!) topic typically omitted in other database texts. Note that the availability of relational comparisons makes the "complicated" operator DIVIDEBY logically unnecessary (forward pointer to Chapter 7). Mention the IS_EMPTY shorthand (it *is* shorthand; to be specific, IS_EMPTY(*r*) is shorthand for *r*{} = TABLE_DUM).

Relational comparisons aren't *relational* operators, since they return a truth value, not a relation.

Explain "*t* ε *r*" and TUPLE FROM *r* (also not relational operators). Don't bother with type inference (here or anywhere else in this chapter).

You've probably already discussed ORDER BY——but if not, then certainly discuss it here.

6.5 Relation Variables

Remind students what a relvar is (relations *vs.* relvars is an important special case of values *vs.* variables in general). We distinguish base relvars *vs.* views ("real *vs.* virtual relvars" in *The Third Manifesto*). Here we're primarily concerned with *base* relvars, but anything we say about "relvars" without that "base" qualifier is true of relvars in general, not just base ones. Remind students that base relvars are **not** necessarily physically stored! To be more specific, the degree of variation allowed between base and stored relvars should be *at least as great* as that allowed between views and base relvars (see Chapter 10); the only logical requirement is that it must be possible to *obtain* the base relvars somehow from those that are physically stored (and then the derived ones can be obtained too). Possible forward pointer to Appendix A?

Explain base relvar definition syntax (and cover *default values* briefly). The terms *heading, body, attribute, tuple, degree,* etc., are all interpreted in the obvious way to apply to relvars as well as relations. Candidate keys and foreign keys will be discussed in detail in Chapter 9. *Note:* Prior to Chapter 9, the book assumes for simplicity that each base relvar has exactly one candidate key, called the **primary** key. In Chapter 9, we're going to argue that the historical emphasis on primary keys

has always been a little bit off base, but don't get into that discussion here.

Relvars have *predicates* (also discussed in Chapter 9).

Explain **relational assignment** (including a reminder re *multiple* assignment) and INSERT, DELETE, and UPDATE shorthands (including **Tutorial D** expansions). Further points to emphasize:

- Remind students re the use of WITH.

- Relational assignment, and hence INSERT, UPDATE, and DELETE, are all **set-level** operations. These operations sometimes can't be simulated by a sequence of tuple-level operations (in fact, there are *no* tuple-level operations in the relational model——one of several reasons why SQL's cursor operations are a bad idea, incidentally).

- Of course, sets sometimes have cardinality one, but updating a set containing just one tuple isn't always possible (assuming the system supports integrity constraints properly——but most don't). See Chapter 9 for further discussion.

- Expressions such as (e.g.) "updating a tuple" are really rather sloppy (though convenient); tuples, like relations, are *values* and *can't* be updated, by definition (quite apart from the fact that we should really be talking about the *set* that contains the tuple in question anyway, instead of about the tuple itself).

Relvars and Their Interpretation

Although not new, this stuff is important and bears repeating. Explain **intended interpretation** and the **Closed World Assumption**. Forward reference to Chapter 9.

6.6 SQL Facilities

SQL supports rows, not tuples (remind students of [some of] the differences). Briefly explain columns, fields, row value constructors, row type constructors, row assignment, row comparisons. *Note:* As a practical matter, nobody——no SQL vendor, that is——supports rows (apart from rows within tables) at the time of writing.

SQL supports tables, not relations (remind students of [some of] the differences, or at least of the fact that they *are*

different). Explain table value constructors. SQL does not support (a) "table type constructors," (b) table assignment, or (c) table comparisons. (It does support IS_EMPTY, more or less, via NOT EXISTS.) Explain the IN operator and "row subqueries" (this term isn't used in the book, but it means a table expression enclosed in parentheses that is required to evaluate to a table containing just one row ... note the coercion involved here!). SQL doesn't properly distinguish between table values and table variables.

Discuss CREATE TABLE* (classic version——we'll get to "typed tables" in a little while). No table-valued columns. Mention DROP and ALTER TABLE if you like.

———————

* Note that "TABLE" in this context means a ***base*** table specifically: a prime indicator of SQL's lack of understanding of relational concepts right there!

———————

The SQL INSERT, UPDATE, and DELETE operations were covered in Chapter 4. SELECT will be covered in more detail in Chapter 8.

There's more to say regarding CREATE TABLE. Recall ***structured types*** from Chapter 5. In that chapter we implied that such types were scalar——though the availability of SQL's "observer and mutator methods" mean they aren't ***really*** scalar, because those methods "break encapsulation" for those structured types (in fact, structured types are more like tuple types in some ways). And——following on from this observation——***such types can be used as the basis for creating base tables:*** The attributes of the structured type become columns of the base table. (Actually the base table has one extra column too, which we'll get to in a moment.) Here's the example from the book:

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL
       REF IS SYSTEM GENERATED ;

CREATE TABLE POINTS OF POINT
     ( REF IS POINT# SYSTEM GENERATED ... ) ;
```

Follow the explanation as given in the book ***but no further*** (more details will come at more appropriate points later). What's this stuff all about? Well, it has to do primarily with the idea of incorporating some kind of "object functionality" into SQL; that's why we defer detailed discussion for now (we need to talk about "objects" in some detail first). But there's nothing in the

standard to say that the features in question can be used only in connection with that object functionality, which is why we at least mention them here. We'll ignore them from this point on, however, until much later (Chapter 26).

References and Bibliography

Reference [6.1] (either version) is strongly recommended and should be distributed to students if at all possible. By contrast, reference [6.2] is mentioned in the book only because it would be inappropriate not to! Students should be warned that few authorities agree with all——or even very many——of the positions articulated in reference [6.2]. See references [6.7] and [6.8] for some specific criticisms.

Answers to Exercises

6.1 Fundamentally, cardinality is a concept that applies to *sets:* The cardinality of a set is the number of elements it contains. However, the concept is extended to other kinds of "collections" also; thus, we speak of the cardinality of a *bag,* the cardinality of a *list,* and so on. In particular, the cardinality of a *relation* is the number of tuples in the body of that relation, and the cardinality of a *relvar* is the cardinality of the relation that happens to be the current value of that relvar. Sometimes the term is even applied to an attribute of some relation, in which case it means the cardinality of either (a) the *bag* or (b) the *set* of values (with duplicates eliminated) appearing in that attribute in that relation. *Note:* Since interpretation (a) is guaranteed to give a result identical to the cardinality of the containing relation, interpretation (b) is probably more common——but watch out for the possibility of confusion in this regard (especially since, to repeat, cardinality is fundamentally a concept that applies to sets rather than bags).

6.2 See Sections 6.2 and 6.3.

6.3 Note first that two *x*'s are equal if and only if they are the same *x,* and this observation is valid regardless of whether the *x*'s are tuples, or tuple types, or relations, or relation types (or anything else).* For tuples, see Section 6.2, subsection "Operators on Tuples." For tuple types, see Section 6.2, subsection "The TUPLE Type Generator." For relations, see Section 6.4, subsection "Operators on Relations." For relation types, see Section 6.3, subsection "The RELATION Type Generator."

———————

Copyright (c) 2003 C. J. Date

* We refer here to what might be called *genuine* equality, not the rather strange kind of equality supported by SQL.

6.4 Predicates:

- S: *Supplier S# is under contract, is named SNAME, has status STATUS, and is located in city CITY.*
- P: *Part P# is of interest,* * *is named PNAME, has color COLOR and weight WEIGHT, and is stored in a warehouse in city CITY.*
- J: *Project J# is under way, is named JNAME, and is located in city CITY.*
- SPJ: *Supplier S# supplies part P# to project J# in quantity QTY.*

---

* For some unspecified reason!

Tutorial D definitions:

```
VAR S BASE RELATION
     { S#      S#,
       SNAME   NAME,
       STATUS  INTEGER,
       CITY    CHAR }
     PRIMARY KEY { S# } ;

VAR P BASE RELATION
     { P#      P#,
       PNAME   NAME,
       COLOR   COLOR,
       WEIGHT  WEIGHT,
       CITY    CHAR }
     PRIMARY KEY { P# } ;

VAR J BASE RELATION
     { J#      J#,
       JNAME   NAME,
       CITY    CHAR }
```

```
        PRIMARY KEY { J# } ;

    VAR SPJ BASE RELATION
          { S#      S#,
            P#      P#,
            J#      J#,
            QTY     QTY }
          PRIMARY KEY { S#, P#, J# }
          FOREIGN KEY { S# } REFERENCES S
          FOREIGN KEY { P# } REFERENCES P
          FOREIGN KEY { J# } REFERENCES J ;
```

6.5
```
TUPLE { S# S# ('S1'), SNAME NAME ('Smith'),
        STATUS 20, CITY 'London' }

TUPLE { P# P# ('P1'), PNAME NAME ('Nut'), COLOR COLOR ('Red'),
        WEIGHT WEIGHT (12.0), CITY 'London' }

TUPLE { J# J# ('J1'), JNAME NAME ('Sorter'), CITY 'Paris' }

TUPLE { S# S# ('S1'), P# P# ('P1'), J# J# ('J1'),
                                    QTY QTY (200) }
```

Of course, no significance attaches to the order in which the arguments appear in any given tuple selector invocation.

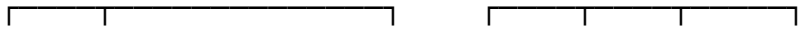
6.6
```
VAR SPJV TUPLE { S# S#, P# P#, J# J#, QTY QTY } ;
```

6.7 They're all relation selector invocations, and they denote, respectively, (a) an empty relation of the same type as relvar SPJ; (b) a relation of the same type as relvar SPJ containing just one tuple (with S# S1, P# P1, J# J1, and QTY 200); (c) a nullary relation containing just one tuple, or in other words TABLE_DEE; (d) same as (c); (e) TABLE_DUM.

6.8 The term has lost much of its original meaning. Originally it meant a relation in which every attribute value is "atomic"——implying that a relation in which some attribute value isn't "atomic" isn't in first normal form. However, we now believe that the term "atomic" has no absolute meaning (in particular, we do ***not*** equate it with **scalar**), and we therefore reject the "atomicity requirement." As far as the relational model is concerned, therefore, ***all*** relations are in first normal form.

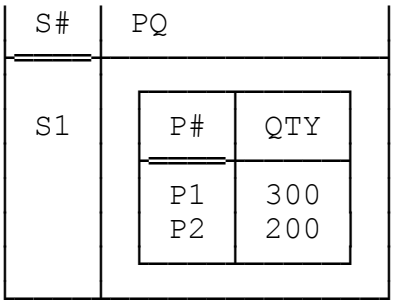
6.9 See Section 6.3, subsection "Relations ***vs***. Tables."

6.10 Here are some possibilities:



Copyright (c) 2003 C. J. Date

a.

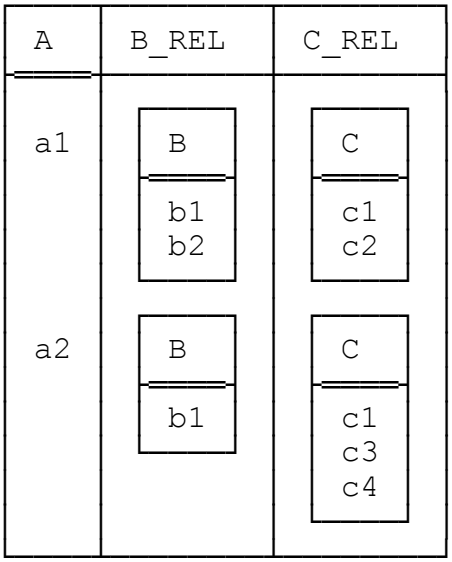


S#	PQ
S1	P# / QTY P1 / 300 P2 / 200

S#	P#	QTY
S1	P1	300
S1	P2	200

Note, however, that a relation like the one on the left can represent a supplier who supplies no parts, while a relation like the one on the right can't.

b.



A	B_REL	C_REL
a1	B: b1, b2	C: c1, c2
a2	B: b1	C: c1, c3, c4

A	B	C
a1	b1	c1
a1	b1	c2
a1	b2	c1
a1	b2	c2
a2	b1	c1
a2	b1	c3
a2	b1	c4

A more concrete example resembling this second pair of relations will be discussed in Chapter 13.

6.11 P{} = TABLE_DUM. ***Explanation:*** The left comparand here is the projection of P on no attributes at all. That projection will yield TABLE_DEE if P currently contains at least one tuple, TABLE_DUM otherwise.

6.12 See Section 6.4, subsections "Operators on Relations."

6.13 If tuple ***t*** satisfies the predicate for relvar ***R*** but doesn't currently appear in ***R,*** then the proposition represented by ***t*** is assumed to be currently false. See Chapter 9 for further explanation of the term "the predicate for relvar ***R***."

6.14 We might agree that a tuple does resemble a ***record*** (occurrence, not type) and an attribute a ***field*** (type, not occurrence). These correspondences are only approximate, however.

Copyright (c) 2003 C. J. Date

A relvar shouldn't be regarded as "just a file," but rather as a *disciplined* file. The discipline in question is one that results in a considerable simplification in the structure of the data as seen by the user, and hence in a corresponding simplification in the operators needed to deal with that data, and indeed in the user interface in general.

6.15

a. INSERT SPJ RELATION { TUPLE { S# S#('S1'), P# P#('P1'),
J# J#('J2'), QTY QTY(500) } } ;

b. INSERT S RELATION { TUPLE { S# S#('S10'), SNAME NAME('Smith'),
CITY 'New York' } } ;

The status for the new supplier will be set to the applicable default value, if there is one; otherwise (i.e., if STATUS has "defaults not allowed"), the INSERT will fail. Note that this error (if it is an error) can be caught at compile time.

c. DELETE P WHERE COLOR = COLOR('Blue') ;

d. DELETE J WHERE IS_EMPTY (((J JOIN SPJ) RENAME J# AS X)
WHERE X = J#) ;

This solution relies on the RENAME operator, to be discussed in Chapter 7.

e. UPDATE P WHERE COLOR = COLOR('Red')
{ COLOR := COLOR('Orange') } ;

f. UPDATE SPJ WHERE S# = S#('S1') { S# := S#('S9') } ,
UPDATE S WHERE S# = S#('S1') { S# := S#('S9') } ;

Note the need to use multiple assignment here (if we used two separate UPDATE statements, a foreign key integrity violation would occur).

6.16 In principle, the answer is yes, it *might* be possible to update the catalog by means of regular INSERT, DELETE, and UPDATE operations. However, allowing such operations would potentially be very dangerous, because it would be all too easy to destroy (inadvertently or otherwise) catalog information that the system needs in order to be able to function correctly. Suppose, for example, that the DELETE operation

DELETE RELVAR WHERE RVNAME = NAME ('SP') ;

(where RELVAR is the catalog relvar that describes the relvars in the database and RVNAME is the attribute in RELVAR that contains relvar names) were allowed on the suppliers-and-parts catalog.

Copyright (c) 2003 C. J. Date

Its effect would be to remove the tuple describing relvar SP from the RELVAR relvar. *As far as the system is concerned, relvar SP would now no longer exist*——i.e., the system would no longer have any knowledge of that relvar. Thus, all subsequent attempts to access that relvar would fail.

In most real products, therefore, INSERT, DELETE, and UPDATE operations on the catalog *either* (a) aren't permitted at all (the usual case) *or* (b) are permitted only to very highly authorized users (perhaps only to the DBA); instead, catalog updates are performed by means of *data definition* statements. For example, defining relvar SP causes (a) an entry to be made for SP in the RELVAR relvar and (b) a set of three entries, one for each of the three attributes of SP, to be made in the ATTRIBUTE relvar (say).* Thus, defining a new object——e.g., a new type, a new operator, or a new base relvar——is in some ways the analog of INSERT for the catalog. Likewise, DROP is the analog of DELETE; and in SQL, which provides a variety of ALTER statements——e.g., ALTER (base) TABLE——for changing catalog entries in various ways, ALTER is the analog of UPDATE.

_____

* It also causes a number of other things to happen that are of no concern to us here.

_____

Note: The catalog also includes entries for the catalog relvars themselves, as we've seen. However, those entries aren't created by explicit data definition operations. Instead, they're created automatically by the system itself as part of the system installation process; in effect, they're "hardwired" into the system.

6.17 There are at least two exceptions. First, relations in the database can't have an attribute of type *pointer*. Second, a relation of type *RT* can't have an attribute of type *RT*. *Note:* The second exception generalizes in an obvious way; for example, a relation of type *RT* can't have an attribute of some relation type *RT'* that in turn has an attribute of type *RT* (and so on).

6.18 A *column* is a component of a table. (Also, SQL often speaks of columns of a *row,* when the row in question is one that's directly contained in a table.) A *field* is a component of a row that isn't (the row, that is) directly contained in a table. An *attribute* is a component of a structured type, or a component of a value or variable of some structured type (however, if a *table* is

defined to be "of" some structured type, then those components are called columns, not attributes). Hmmm ...

6.19 The change causes table POINTS, and (typically) applications that use that table, to "break." Lack of data independence!

***** End of Chapter 6 *****

Chapter 7

R e l a t i o n a l A l g e b r a

Principal Sections

- Closure revisited
- Syntax
- Semantics
- Examples
- What's the algebra for?
- Further points
- Additional operators
- Grouping and ungrouping

General Remarks

No "SQL Facilities" section in this chapter——it's deferred to Chapter 8, for reasons to be explained in that chapter. There are, however, many references to SQL in this chapter in passing.

Begin with a quick overview of "the original eight operators" (Fig. 7.1, repeated for convenience on the left endpaper at the back of the book). A small point: What I'm calling the "original" algebra is not quite the same as the set of operators defined in Codd's original relational model paper [6.1]. See Chapter 2 of reference [6.9] for a detailed discussion of the operators from reference [6.1].

Stress the point that the relational algebra——or, equivalently, the relational calculus——is ***part of the relational model***. Some writers seem not to understand this point! For example, one textbook has a chapter entitled "The Relational Data Model and Relational Algebra," and another has separate chapters entitled "The Relational Model" and "Relational Algebra and Calculus." Perhaps the confusion arises because of the secondary meaning of the term ***data model*** as a model of the persistent data of some particular enterprise, where the manipulative aspects are very much downplayed, or even ignored altogether.

Note: According to ***Chambers Twentieth Century Dictionary,*** an ***algebra*** is "[a system] using symbols and involving reasoning about relationships and operations." (Math texts offer much more precise definitions, of course, but this one is good enough for our purposes.) More specifically, an algebra consists of a set of ***objects*** and a set of ***operators*** that together satisfy certain

Copyright (c) 2003 C. J. Date

axioms or *laws,* such as the laws of closure, commutativity, associativity, and so on (closure is particularly important, of course). The word "algebra" itself ultimately derives from Arabic *al-jebr,* meaning a *resetting* (of something broken) or a *combination*.

The operators are (a) *generic,* (b) *read-only*.

Finally, please note the following remarks from near the end of Section 7.1 (slightly reworded here):

(Begin quote)

We often talk about, e.g., "the projection over attribute *A* of relvar *R*," meaning the relation that results from taking the projection over that attribute *A* of the current value of that relvar *R*. Occasionally, however, it's convenient to use expressions like "the projection over attribute *A* of relvar *R*" in a slightly different sense. For example, suppose we define a view SC of the suppliers relvar S that consists of just the S# and CITY attributes of that relvar. Then we might say, loosely but very conveniently, that relvar SC is "the projection over S# and CITY of relvar S"——meaning, more precisely, that the value of SC at any given time is the projection over S# and CITY of the value of relvar S at that time. In a sense, therefore, we can talk in terms of projections of relvars *per se,* rather than just in terms of projections of current values of relvars. We hope this kind of dual usage of the terminology on our part does not cause any confusion.

(End quote)

The foregoing remarks are particularly pertinent to discussions of *views* (Chapter 10) and *dependencies and further normalization* (Chapters 11-13).

7.2 Closure Revisited

The emphasis in this section on **relation type inference rules,** and the consequent need for a (column) **RENAME** operator, are further features that distinguish this book from its competitors. (The need for such rules was first noted at least as far back as 1975 [7.10], but they still get little play in the literature.) Note too the concomitant requirement that (where applicable) operators be defined in terms of *matching attributes;* e.g., JOIN requires the joining attributes to have the same name——as well as the same type, of course.* SQL doesn't work this way, and nor does the relational algebra as described in most of the literature; after much investigation into (and experimentation with) other

approaches, however, I believe strongly that this scheme is the best basis on which to build and move forward.

---

* In other words, each joining attribute is ***the same attribute*** in the two relations (see the remarks on this topic in the previous chapter of this manual).

Stress the points that (a) RENAME is not like SQL's ALTER TABLE, (b) a RENAME invocation is an expression, not a command or statement (so it can be nested inside other expressions).

By the way, it's worth noting that, in a sense, the relational algebra is "**more** closed" than ordinary arithmetic, inasmuch as it includes nothing analogous to the "divide by zero" problem in arithmetic (TABLE_DUM and TABLE_DEE are relevant here!). See Exercise 7.9.

7.3 Syntax / 7.4 Semantics / 7.5 Examples

These sections should be mostly self-explanatory. Just a few points:

- Codd had a very specific purpose in mind, which we'll examine in the next chapter, for defining just the eight operators he did. But any number of operators can be defined that satisfy the simple requirement of "relations in, relations out," and many additional operators have indeed been defined, by many different writers. We'll discuss the original eight first——not exactly as they were originally defined but as they've since become——and use them as the basis for discussing a variety of algebraic ideas; then we'll go on to consider some of the many useful operators that have subsequently been added to the original set.

- Remind students that most of these operators rely on tuple equality for their definition (give one or two examples).

- Regarding union, intersection, and difference, you might be interested to note that an extensive discussion of the troubles that plague SQL in connection with these operators can be found in the article "A Sweet Disorder," already mentioned in Chapters 4 and 6 of this manual.

- Note the generalization of the restrict operator.

- Stress the fact that joins are *not* always between a foreign key and a matching primary (or candidate) key. *Note:* Candidate keys were first briefly mentioned in Chapter 6 but won't be fully explained until Chapter 9.

- Note that the book correctly defines TIMES as a degenerate case of JOIN. By contrast, other presentations of the algebra usually——also correctly, but less desirably——define JOIN in terms of TIMES (i.e., as a projection of a restriction of a product). See Exercise 7.5.

- You might want to skip divide, since relational comparisons do the job better (it might be sufficient just to mention that very point). If you do cover it, however, note that Codd's original divide [7.1] was a dyadic operator; the "Small Divide," by contrast, is a triadic one. Consider the query "Get supplier numbers for suppliers who supply all purple parts." A putative formulation of this query, using Codd's divide (see the annotation to reference [7.4]), might look like this:

```
SP { S#, P# } DIVIDEBY
              ( P WHERE COLOR = COLOR ('Purple') ) { P# }
```

  This formulation is incorrect, however. Suppose there are no purple parts. Then every supplier supplies all of them!——even suppliers like supplier S5 who supply no parts at all (given our usual sample data values). Yet the formulation shown can't possibly return suppliers who supply no parts at all, because such suppliers aren't represented in SP in the first place.

  Note: If you're having difficulty with the idea that supplier S5 supplies all purple parts, consider the statement: "For all purple parts *p*, supplier S5 supplies part *p*." This statement in turn is logically equivalent to: "There does not exist a purple part *p* such that supplier S5 does not supply *p*." And this latter statement undeniably evaluates to TRUE, because the opening quantified expression "There does not exist a purple part *p*" certainly evaluates to TRUE. (More generally, the expression NOT EXISTS *x* (...) certainly evaluates to TRUE——**regardless of what the "..." stands for**——if there aren't any *x*'s.)

  A correct formulation of the query, using the Small Divide, looks like this:

```
S { S# } DIVIDEBY
```

```
( P WHERE COLOR = COLOR ('Purple') ) { P# }
                                    PER SP { S#, P# }
```

- Again stress the usefulness of WITH in breaking complex expressions down into "step-at-a-time" ones. Also stress the fact that using WITH does *not* sacrifice nonprocedurality.

Note: The discussion of projection in Section 7.4 includes the following question: Why can't any attribute be mentioned more than once in the attribute name commalist? The answer, of course, is that the commalist is supposed to denote a *set* of attributes, and attribute names in the result must therefore be unique.

The discussion under Example 7.5.5 includes the following text:

(Begin quote)

The purpose of the condition SA < SB is twofold:

- It eliminates pairs of supplier numbers of the form (*x,x*).
- It guarantees that the pairs (*x,y*) and (*y,x*) won't both appear.

(End quote)

The example might thus be used, if desired, to introduce the concepts of:

- *Reflexivity:* A binary relation *R*{*A,B*} is reflexive if and only if *A* and *B* are of the same type and the tuple {*A:x,B:x*} appears in *R* for all applicable values *x* of that common type.
- *Symmetry:* A binary relation *R*{*A,B*} is reflexive if and only if *A* and *B* are of the same type and whenever the tuple {*A:x,B:y*} appears in *R*, then the tuple {*A:y,B:x*} also appears in *R*).

See, e.g., reference [24.1] for further discussion.

7.6 What's the Algebra for?

Dispel the popular misconception that the algebra (or the calculus) is just for queries. Note in particular that the algebra or calculus is fundamentally required in order to be able to express *integrity constraints*, which is why Chapters 7 and 8 precede Chapter 9.

Copyright (c) 2003 C. J. Date

Regarding ***relational completeness:*** The point is worth making that, once Codd had defined this notion of linguistic power, it really became incumbent on the designer of any database language ***either*** to ensure that the language in question was at least that powerful ***or*** to have a really good justification for not doing so. And there really isn't any good justification ... This fact is a cogent criticism of several nonrelational database languages, including object ones in particular and, I strongly suspect, XML query languages (see Chapter 27).

Regarding ***primitive operators:*** The "RISC algebra" **A** is worth a mention.

The section includes the following inline exercise: The expression

```
( ( SP JOIN S ) WHERE P# = P# ( 'P2' ) ) { SNAME }
```

can be transformed into the logically equivalent, but probably more efficient, expression

```
( ( SP WHERE P# = P# ( 'P2' ) ) JOIN S ) { SNAME }
```

In what sense is the second expression probably more efficient? Why only "probably"?

Answer: The second expression performs the restriction before the join. Loosely speaking, therefore, it reduces the size of the input to the join, meaning there's less data to be scanned to do the join, and the result of the join is smaller as well. In fact, the second expression might allow the result of the join to be kept in main memory, while the first might not; thus, there could be orders of magnitude difference in performance between the two expressions.

On the other hand, recall that the relational model has nothing to say regarding physical storage. Thus, for example, the join of SP and S might be physically stored as a single file, in which case the first expression might perform better. Also, there will be little performance difference between the two expressions anyway if the relations are small (as an extreme case, consider what happens if they're both empty).

This might be a good place to digress for a couple of minutes to explain why duplicate tuples inhibit optimization! A detailed example and discussion can be found in reference [6.6]. That same paper also refutes the claim that a "tuple-bag algebra" is "just as respectable" (and in particular just as optimizable) as the relational algebra.

7.7 Further Points

Explain associativity and commutativity briefly and show which operators are associative and which commutative. Discuss some of the implications. ***Note:*** One such implication, not explicitly mentioned in the book, is that we can legitimately talk about (e.g.) **the** join of any number of relations (i.e., such an expression does have a well-defined unique meaning).

Also explain the specified equivalences——especially the ones involving TABLE_DEE. Introduce the terms "identity restriction," etc.

Define "joins" (etc.) of one relation and of no relations at all. I wouldn't bother to get into the specifics of why the join of no relations and the intersection of no relations aren't the same! But if you're interested, an explanation can be found in Chapter 1 of reference [23.4]. See also Exercise 7.10.

7.8 Additional Operators

Regarding ***semijoin:*** It's worth noting that semijoin is often more directly useful in practice than join is! A typical query is "Get suppliers who supply part P2." Using SEMIJOIN:

```
S SEMIJOIN ( SP WHERE P# = P# ( 'P2' ) )
```

Without SEMIJOIN:

```
( S JOIN ( SP WHERE P# = P# ( 'P2' ) ) )
                              { S#, SNAME, STATUS, CITY }
```

It might be helpful to point out that the SQL analog refers to table S (only) in the SELECT and FROM clauses and mentions table SP only in the WHERE clause:

```
SELECT *
FROM   S
WHERE  S# IN
     ( SELECT S#
       FROM   SP
       WHERE  P# = 'P2' ) ;
```

In a sense, this SQL expression corresponds more directly to the semijoin formulation than to the join one.

Analogous remarks apply to ***semidifference***.

Regarding ***extend:*** EXTEND is one of the most useful operators of all. Consider the query "Get parts and their weight in grams for parts whose gram weight exceeds 10000" (recall that part weights are given in pounds). Relational algebra formulation:*

```
( EXTEND P ADD ( WEIGHT * 454 ) AS GMWT )
           WHERE GMWT > WEIGHT ( 10000.0 )
```

Conventional SQL analog (note the repeated subexpression):

```
SELECT P.*, ( WEIGHT * 454 ) AS GMWT
FROM   P
WHERE  ( WEIGHT * 454 ) > 10000.0 ;
```

The name GMWT cannot be used in the WHERE clause because it's the name of a column of the ***result*** table.

---

* The discussion of EXTEND in the book asks what the type of the result of the expression WEIGHT * 454 is. As this formulation suggests, the answer is, obviously enough, WEIGHT once again. However, if we assume (as we're supposed to) that WEIGHT values are given in pounds, then the result of WEIGHT * 454 presumably has to be interpreted as a weight in pounds, too!——not as a weight in grams. Clearly something strange is going on here ... See the discussion of ***units of measure*** in Chapter 5, Section 5.4.

As this example suggests, the SQL idea that all queries must be expressed as a projection (SELECT) of a restriction (WHERE) of a product (FROM) is really much too rigid, and of course there's no such limitation in the relational algebra——operations can be combined in arbitrary ways and executed in arbitrary sequences.

Note: It's true that the SQL standard would now allow the repetition of the subexpression to be avoided as follows:

```
SELECT P#, PNAME, COLOR, WEIGHT, CITY, GMWT
FROM ( SELECT P.*, ( WEIGHT * 454 ) AS GMWT
       FROM   P ) AS POINTLESS
WHERE  GMWT > 10000.0 ;
```

(The specification AS ***POINTLESS*** is pointless but is required by SQL's syntax rules——see reference [4.20].) However, not all SQL products permit subqueries in the FROM clause at the time of writing. Note too that a select-item of the form "P.*" in the

outer SELECT clause would be illegal in this formulation! See reference [4.20] for further discussion of this point also.

Note: The subsection on EXTEND is also the place where the *aggregate operators* COUNT, SUM, etc., are first mentioned. Observe the important differences (both syntactic and semantic) in the treatment of such operators between **Tutorial D** and SQL. Note too the aggregate operators ALL and ANY, both of which operate on arguments consisting of boolean values; ALL returns TRUE if and only if all arguments evaluate to TRUE, ANY returns TRUE if and only if any argument does.

Regarding *summarize:* As the book says, please note that a *<summarize add>* is *not* the same thing as an *<aggregate operator invocation>*. An *<aggregate operator invocation>* is a scalar expression and can appear wherever a scalar selector invocation——in particular, a scalar literal——can appear. A *<summarize add>,* by contrast, is merely a SUMMARIZE operand; it's *not* a scalar expression, it has no meaning outside the context of SUMMARIZE, and in fact it can't appear outside that context.

Note the two forms of SUMMARIZE (PER and BY).

Regarding *tclose:* Don't go into much detail. The operator is mentioned here mainly for completeness. Do note, though, that it really is a new primitive——it can't be defined in terms of operators we've already discussed. (Explain why? See the answer to Exercise 8.7 in the next chapter.)

7.8 Grouping and Ungrouping

This section could either be deferred or assigned as background reading.* Certainly the remarks on reversibility shouldn't be gone into too closely on a first pass. Perhaps just say that since we allow relation-valued attributes, we need a way of mapping between relations with such attributes and relations without them, and that's what GROUP and UNGROUP are for. Show an ungrouped relation and its grouped counterpart; that's probably sufficient.

———————

* The article "What Does First Normal Form Really Mean?" (already mentioned in Chapter 6 of this manual) is relevant.

———————

Note clearly that "grouping" as described here is ***not*** the same thing as the GROUP BY operation in SQL——it returns a ***relation*** (with a relation-valued attribute), not an SQL-style "grouped table." In fact, SQL's GROUP BY violates the relational closure property.

Relations with relation-valued attributes are ***not*** "NF² relations"! In fact, it's hard to say exactly what "NF² relations" ***are***——the concept doesn't seem too coherent when you really poke into it. (Certainly we don't need all of the additional operators——and additional complexity——that "NF² relations" seem to involve.)

Answers to Exercises

7.1 The only operators whose definitions don't rely on tuple equality are restrict, Cartesian product, extend, and ungroup. (Even these cases are debatable, as a matter of fact.)

7.2 The trap is that the join involves the CITY attributes as well as the S# and P# attributes. The result looks like this:

S#	SNAME	STATUS	CITY	P#	QTY	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	300	Nut	Red	12.0
S1	Smith	20	London	P4	200	Screw	Red	14.0
S1	Smith	20	London	P6	100	Cog	Red	19.0
S2	Jones	10	Paris	P2	400	Bolt	Green	17.0
S3	Blake	30	Paris	P2	200	Bolt	Green	17.0
S4	Clark	20	London	P4	200	Screw	Red	14.0

7.3 2^n. This count includes the ***identity*** projection (i.e., the projection over all n attributes), which yields a result identical to the original relation r, and the ***nullary*** projection (i.e., the projection over no attributes at all), which yields TABLE_DUM if the original relation r is empty and TABLE_DEE otherwise.

7.4 INTERSECT and TIMES are both special cases of JOIN, so we can ignore them here. The ***commutativity*** of UNION and JOIN is obvious from the definitions, which are symmetric in the two relations concerned. We can show that UNION is ***associative*** as follows. Let t be a tuple. Then:*

t ε A UNION (B UNION C) iff t ε A OR t ε (B UNION C),
i.e., iff t ε A OR (t ε B OR t ε C),
i.e., iff (t ε A OR t ε B) OR t ε C,

i.e., iff *t* ε (A UNION B) OR *t* ε C,
i.e., iff *t* ε (A UNION B) UNION C.

Note the appeal in the third line to the associativity of OR.

---

* The shorthand "iff" stands for "if and only if."

The proof that JOIN is associative is analogous.

7.5 We omit the verifications, which are straightforward. The answer to the last part of the exercise is *b* SEMIJOIN *a*.

7.6 JOIN is discussed in Section 7.4. INTERSECT can be defined as follows:

A INTERSECT *B* ≡ *A* MINUS (*A* MINUS *B*)

or (equally well)

A INTERSECT *B* ≡ *B* MINUS (*B* MINUS *A*)

These equivalences, though valid, are slightly unsatisfactory, since *A* INTERSECT *B* is symmetric in *A* and *B* and the other two expressions aren't. Here by contrast is a symmetric equivalent:

(*A* MINUS (*A* MINUS *B*)) UNION (*B* MINUS (*B* MINUS *A*))

Note: Given that *A* and *B* must be of the same type, we also have:

A INTERSECT *B* ≡ *A* JOIN *B*

As for DIVIDEBY, we have:

```
A DIVIDEBY B PER C  ≡  A { X }
                         MINUS ( ( A { X } TIMES B { Y } )
                                       MINUS C { X, Y } ) { X }
```

Here *X* is the set of attributes common to *A* and *C* and *Y* is the set of attributes common to *B* and *C*.

Note: DIVIDEBY as just defined is actually a generalization of the version defined in the body of the chapter——though it's still a Small Divide [7.4]——inasmuch as we assumed previously that *A* had no attributes apart from *X*, *B* had no attributes apart from

Y, and *C* had no attributes apart from *X* and *Y*. The foregoing generalization would allow, e.g., the query "Get supplier numbers for suppliers who supply all parts," to be expressed more simply as just

```
S DIVIDEBY P PER SP
```

instead of (as previously) as

```
S { S# } DIVIDEBY P { P# } PER SP { S#, P# }
```

7.7 The short answer is no. Codd's original DIVIDEBY did satisfy the property that

(*a* TIMES *b*) DIVIDEBY *b* ≡ *a*

so long as *b* is nonempty (what happens otherwise?). However:

- Codd's DIVIDEBY was a dyadic operator; our DIVIDEBY is triadic, and hence can't possibly satisfy a similar property.

- In any case, even with Codd's DIVIDEBY, dividing *a* by *b* and then forming the Cartesian product of the result with *b* will yield a relation that ***might*** be identical to *a*, but is more likely to be some proper subset of *a*:

  (*A* DIVIDEBY *B*) TIMES *B* ⊆ *A*

  Codd's DIVIDEBY is thus more analogous to ***integer*** division in ordinary arithmetic (i.e., it ignores the remainder).

7.8 We can say that TABLE_DEE (DEE for short) is the analog of 1 with respect to multiplication in ordinary arithmetic because

r TIMES DEE ≡ DEE TIMES *r* ≡ *r*

for all relations *r* (in other words, DEE is the **identity** with respect to TIMES and, more generally, with respect to JOIN). However, there's no relation that behaves with respect to TIMES in a way that is ***exactly*** analogous to the way that 0 behaves with respect to multiplication——but the behavior of TABLE_DUM (DUM for short) is somewhat reminiscent of the behavior of 0, inasmuch as

r TIMES DUM ≡ DUM TIMES *r* ≡ an empty relation with the same heading as *r*

for all relations *r*.

We turn now to the effect of the algebraic operators on DEE and DUM. We note first that the only relations that are of the same type as DEE and DUM are DEE and DUM themselves. We have:

UNION	DEE	DUM
DEE	DEE	DEE
DUM	DEE	DUM

INTERSECT	DEE	DUM
DEE	DEE	DUM
DUM	DUM	DUM

MINUS	DEE	DUM
DEE	DUM	DEE
DUM	DUM	DUM

In the case of MINUS, the first operand is shown at the left and the second at the top (for the other operators, of course, the operands are interchangeable). Notice how reminiscent these tables are of the truth tables for OR, AND, and AND NOT, respectively; of course, the resemblance isn't a coincidence.

As for restrict and project, we have:

- Any restriction of DEE yields DEE if the restriction condition evaluates to TRUE, DUM if it evaluates to FALSE.
- Any restriction of DUM yields DUM.
- Projection of any relation over no attributes yields DUM if the original relation is empty, DEE otherwise. In particular, projection of DEE or DUM, necessarily over no attributes at all, returns its input.

For extend and summarize, we have:

- Extending DEE or DUM to add a new attribute yields a relation of degree one and the same cardinality as its input.
- Summarizing DEE or DUM (necessarily by no attributes at all) yields a relation of degree one and the same cardinality as its input.

Note: We omit consideration of DIVIDEBY, SEMIJOIN, and SEMIMINUS because they're not primitive. TCLOSE is irrelevant (it applies to binary relations only). We also omit consideration of GROUP and UNGROUP for obvious reasons.

7.9 No!

7.10 INTERSECT is defined only if its operand relations are all of the same type, while no such limitation applies to JOIN. It follows that, when there are no operands at all, we can define the result for JOIN ***generically,*** but we can't do the same for INTERSECT——we can define the result only for ***specific*** INTERSECT operations (i.e., INTERSECT operations that are specific to some particular relation type). In fact, when we say that INTERSECT is

a special case of JOIN, what we really mean is that every *specific* INTERSECT is a special case of some *specific* JOIN. Let S_JOIN be such a specific JOIN. Then S_JOIN and JOIN aren't the same operator, and it's reasonable to say that the S_JOIN and the JOIN of no relations at all give different results.

7.11 In every case the result is a relation of degree one. If *r* is nonempty, all four expressions return a one-tuple relation containing the cardinality *n* of *r*. If *r* is empty, expressions a. and c. both return an empty result, while expressions b. and d. both return a one-tuple relation containing zero (the cardinality of *r*).

7.12 Relation *r* has the same cardinality as SP and the same heading, except that it has one additional attribute, X, which is relation-valued. The relations that are values of X have degree zero (i.e., they are nullary relations); furthermore, each of those relations is TABLE_DEE, not TABLE_DUM, because every tuple *sp* in SP effectively includes the 0-tuple as its value for that subtuple of *sp* that corresponds to the empty set of attributes. Thus, each tuple in *r* effectively consists of the corresponding tuple from SP extended with the X value TABLE_DEE.

The expression *r* UNGROUP X yields the original SP relation again.

7.13 J

7.14 J WHERE CITY = 'London'

7.15 (SPJ WHERE J# = J# ('J1')) { S# }

7.16 SPJ WHERE QTY ≥ QTY (300) AND QTY ≤ QTY (750)

7.17 P { COLOR, CITY }

7.18 (S JOIN P JOIN J) { S#, P#, J# }

7.19
```
( ( ( S RENAME CITY AS SCITY ) TIMES
    ( P RENAME CITY AS PCITY ) TIMES
    ( J RENAME CITY AS JCITY ) )
  WHERE SCITY =/ PCITY
  OR    PCITY =/ JCITY
  OR    JCITY =/ SCITY ) { S#, P#, J# }
```

7.20
```
( ( ( S RENAME CITY AS SCITY ) TIMES
    ( P RENAME CITY AS PCITY ) TIMES
    ( J RENAME CITY AS JCITY ) )
  WHERE SCITY =/ PCITY
  AND   PCITY =/ JCITY
```

```
     AND   JCITY =/ SCITY ) { S#, P#, J# }
```

7.21
```
P SEMIJOIN ( SPJ SEMIJOIN ( S WHERE CITY = 'London' ) )
```

7.22 Just to remind you of the possibility, we show a step-at-a-time solution to this exercise:

```
     WITH ( S WHERE CITY = 'London' ) AS T1,
          ( J WHERE CITY = 'London' ) AS T2,
          ( SPJ JOIN T1 ) AS T3,
          T3 { P#, J# } AS T4,
          ( T4 JOIN T2 ) AS T5 :
          T5 { P# }
```

Here's the same query without using WITH:

```
     ( ( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P#, J# }
                     JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

We'll give a mixture of solutions (some using WITH, some not) to the remaining exercises.

7.23
```
( ( S RENAME CITY AS SCITY ) JOIN SPJ JOIN
  ( J RENAME CITY AS JCITY ) ) { SCITY, JCITY }
```

7.24
```
( J JOIN SPJ JOIN S ) { P# }
```

7.25
```
( ( ( J RENAME CITY AS JCITY ) JOIN SPJ JOIN
    ( S RENAME CITY AS SCITY ) )
                          WHERE JCITY =/ SCITY ) { J# }
```

7.26
```
WITH ( SPJ { S#, P# } RENAME ( S# AS XS#, P# AS XP# ) )
                                                  AS T1,
     ( SPJ { S#, P# } RENAME ( S# AS YS#, P# AS YP# ) )
                                                  AS T2,
     ( T1 TIMES T2 ) AS T3,
      ( T3 WHERE XS# = YS# AND XP# < YP# ) AS T4 :
     T4 { XP#, YP# }
```

7.27
```
( SUMMARIZE SPJ { S#, J# }
  PER RELATION { TUPLE { S# S# ( 'S1' ) } }
  ADD COUNT AS N ) { N }
```

The expression in the PER clause here is a relation selector invocation (in fact, it's a relation literal, denoting a relation containing just one tuple).

7.28
```
( SUMMARIZE SPJ { S#, P#, QTY }
  PER RELATION { TUPLE { S# S# ( 'S1' ), P# P# ( 'P1' ) } }
  ADD SUM ( QTY ) AS Q ) { Q }
```

7.29
```
SUMMARIZE SPJ PER SPJ { P#, J# } ADD SUM ( QTY ) AS Q
```

7.30
```
WITH ( SUMMARIZE SPJ PER SPJ { P#, J# }
                       ADD AVG ( QTY ) AS Q ) AS T1,
       ( T1 WHERE Q > QTY ( 350 ) ) AS T2 :
       T2 { P# }
```

7.31
```
( J JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) ) { JNAME }
```

7.32
```
( P JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) ) { COLOR }
```

7.33
```
( SPJ JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

7.34
```
( SPJ JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) { P# } ) { J# }
```

7.35
```
( ( ( SPJ JOIN
      ( P WHERE COLOR = COLOR ( 'Red' ) ) { P# } ) { S# }
                       JOIN SPJ ) { P# } JOIN SPJ ) { S# }
```

7.36
```
WITH ( S { S#, STATUS } RENAME ( S# AS XS#,
                                 STATUS AS XSTATUS ) ) AS T1,
       ( S { S#, STATUS } RENAME ( S# AS YS#,
                                 STATUS AS YSTATUS ) ) AS T2,
       ( T1 TIMES T2 ) AS T3,
       ( T3 WHERE XS# = S# ( 'S1' ) AND
                 XSTATUS > YSTATUS ) AS T4 :
       T4 { YS# }
```

7.37
```
( ( EXTEND J ADD MIN ( J, CITY ) AS FIRST )
                                 WHERE CITY = FIRST ) { J# }
```

7.38
```
WITH ( SPJ RENAME J# AS ZJ# ) AS T1,
       ( T1 WHERE ZJ# = J# AND P# = P# ( 'P1' ) ) AS T2,
       ( SPJ WHERE P# = P# ( 'P1' ) ) AS T3,
       ( EXTEND T3 ADD AVG ( T2, QTY ) AS QX ) AS T4,
       T4 { J#, QX } AS T5,
       ( SPJ WHERE J# = J# ( 'J1' ) ) AS T6,
       ( EXTEND T6 ADD MAX ( T6, QTY ) AS QY ) AS T7,
       ( T5 TIMES T7 { QY } ) AS T8,
       ( T8 WHERE QX > QY ) AS T9 :
       T9 { J# }
```

7.39
```
WITH ( SPJ WHERE P# = P# ( 'P1' ) ) AS T1,
       T1 { S#, J#, QTY } AS T2,
        ( T2 RENAME ( J# AS XJ#, QTY AS XQ ) ) AS T3,
       ( SUMMARIZE T1 PER SPJ { J# }
                     ADD AVG ( QTY ) AS Q ) AS T4,
       ( T3 TIMES T4 ) AS T5,
       ( T5 WHERE XJ# = J# AND XQ > Q ) AS T6 :
```

Copyright (c) 2003 C. J. Date

```
          T6 { S# }
```

7.40
```
WITH ( S WHERE CITY = 'London' ) { S# } AS T1,
     ( P WHERE COLOR = COLOR ( 'Red' ) ) AS T2,
     ( T1 JOIN SPJ JOIN T2 ) AS T3 :
     J { J# } MINUS T3 { J# }
```

7.41
```
J { J# } MINUS ( SPJ WHERE S# =/ S# ( 'S1' ) ) { J# }
```

7.42
```
WITH ( ( SPJ RENAME P# AS X ) WHERE X = P# ) { J# } AS T1,
     ( J WHERE CITY = 'London' ) { J# } AS T2,
     ( P WHERE T1 ≥ T2 ) AS T3 :
     T3 { P# }
```

7.43
```
S { S#, P# } DIVIDEBY J { J# } PER SPJ { S#, P#, J# }
```

7.44
```
( J WHERE
    ( ( SPJ RENAME J# AS Y ) WHERE Y = J# ) { P# } ≥
    ( SPJ WHERE S# = S# ( 'S1' ) ) { P# } ) { J# }
```

7.45
```
S { CITY } UNION P { CITY } UNION J { CITY }
```

7.46
```
( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P# }
  UNION
( SPJ JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

7.47
```
( S TIMES P ) { S#, P# } MINUS SP { S#, P# }
```

7.48 We show two solutions to this problem. The first, which is due to Hugh Darwen, uses only the operators of Sections 7.3-7.4:

```
WITH ( SP RENAME S# AS SA ) { SA, P# } AS T1,
     /* T1 {SA,P#} : SA supplies part P# */

     ( SP RENAME S# AS SB ) { SB, P# } AS T2,
     /* T2 {SB,P#} : SB supplies part P# */

     T1 { SA } AS T3,
     /* T3 {SA} : SA supplies some part */

     T2 { SB } AS T4,
     /* T4 {SB} : SB supplies some part */

     ( T1 TIMES T4 ) AS T5,
      /* T5 {SA,SB,P#} : SA supplies some part and
                         SB supplies part P# */

     ( T2 TIMES T3 ) AS T6,
      /* T6 {SA,SB,P#} : SB supplies some part and
                         SA supplies part P# */
```

```
( T1 JOIN T2 ) AS T7,
/* T7 {SA,SB,P#} : SA and SB both supply part P# */

( T3 TIMES T4 ) AS T8,
/* T8 {SA,SB} : SA supplies some part and
                SB supplies some part */

SP { P# } AS T9,
/* T9 {P#} : part P# is supplied by some supplier */

( T8 TIMES T9 ) AS T10,
/* T10 {SA,SB,P#} :
   SA supplies some part,
   SB supplies some part, and
    part P# is supplied by some supplier */

( T10 MINUS T7 ) AS T11,
/* T11 {SA,SB,P#} : part P# is supplied,
                    but not by both SA and SB */

( T6 INTERSECT T11 ) AS T12,
 /* T12 {SA,SB,P#} : part P# is supplied by SA
                     but not by SB */

( T5 INTERSECT T11 ) AS T13,
 /* T13 {SA,SB,P#} : part P# is supplied by SB
                     but not by SA */

T12 { SA, SB } AS T14,
/* T14 {SA,SB} :
   SA supplies some part not supplied by SB */

T13 { SA, SB } AS T15,
/* T15 {SA,SB} :
   SB supplies some part not supplied by SA */

( T14 UNION T15 ) AS T16,
/* T16 {SA,SB} : some part is supplied by SA or SB
                 but not both */

T7 { SA, SB } AS T17,
/* T17 {SA,SB} :
  some part is supplied by both SA and SB */

( T17 MINUS T16 ) AS T18,
/* T18 {SA,SB} :
   some part is supplied by both SA and SB,
   and no part supplied by SA is not supplied by SB,
   and no part supplied by SB is not supplied by SA
   -- so SA and SB each supply exactly the same parts */
```

```
          ( T18 WHERE SA < SB ) AS T19 :
          /* tidy-up step */

     T19
```

The second solution——which is much more straightforward!——makes use of the relational comparisons introduced in Chapter 6:

```
     WITH ( S RENAME S# AS SA ) { SA } AS RA ,
          ( S RENAME S# AS SB ) { SB } AS RB :
          ( RA TIMES RB )
               WHERE ( SP WHERE S# = SA ) { P# } =
                     ( SP WHERE S# = SB ) { P# }
                 AND   SA < SB
```

7.49 SPJ GROUP (J#, QTY) AS JQ

7.50 Let SPQ denote the result of the expression shown in the answer to Exercise 7.49. Then:

```
     SPQ UNGROUP JQ
```

***** End of Chapter 7 *****

Chapter 8

Relational Calculus

Principal Sections

- Tuple calculus
- Examples
- Calculus *vs.* algebra
- Computational capabilities
- SQL facilities
- Domain calculus
- Query-By-Example

General Remarks

As noted in the discussion of the introduction to this part of the book, it might be possible, or even advisable, to skip much of this chapter on a first pass. The SQL stuff probably needs to be covered, though (if you didn't already cover it in Chapter 4). And "database professionals"——i.e., anyone who's serious about the subject of database technology——really ought to be familiar with both tuple and domain calculus. And everybody ought at least to understand the **quantifiers**.

Note: The term "calculus" signifies merely a *system of computation* (the Latin word *calculus* means a pebble, perhaps used in counting or some other form of reckoning). Thus, relational calculus can be thought of as a system for computing with relations. Incidentally, it's common to assert (as Section 8.1 in fact does) that the relational model is based on *predicate* calculus specifically. In a real computer system, however, all domains and relations are necessarily finite, and the predicate calculus thus degenerates——at least in principle——to the simpler *propositional* calculus. In particular, the quantifiers EXISTS and FORALL can therefore be defined (as indeed they are in Section 8.2) as iterated OR and AND, respectively.

A brief overview of Codd's ALPHA language appears in reference [6.9] (Chapters 6 and 7).

8.2 Tuple Calculus

It would be possible to skip the rather formal presentation in this section and go straight to the more intuitively understandable examples in Section 8.3.

This section claims that the abbreviation WFF is pronounced "weff," but the pronunciations "wiff" and "woof" are also heard.

Let *V* range over an empty relation. Then it must be clearly understood that EXISTS *V* (*p*(*V*)) gives FALSE and FORALL *V* (*p*(*V*)) gives TRUE, **regardless of the nature of *p***.

8.3 Examples

This section suggests that algebraic versions of the examples be given as well, for "compare and contrast" purposes. In fact algebraic versions of most of them can be found in Chapter 6. To be specific:

Example 8.3.2 *corresponds to* Example 6.5.5
Example 8.3.3 *corresponds to* Example 6.5.1 (almost)
Example 8.3.4 *corresponds to* Example 6.5.2
Example 8.3.6 *corresponds to* Example 6.5.3
Example 8.3.7 *corresponds to* Example 6.5.6
Example 8.3.8 *corresponds to* Example 6.5.4

Here are algebraic versions of the other three:

- *Example 8.3.1:*

```
( S WHERE CITY = 'Paris' AND STATUS > 20 ) { S#, STATUS }
```

- *Example 8.3.5:*

```
( ( ( SP WHERE S# = S# ( 'S2' ) ) { P# } JOIN SP ) JOIN S )
                                                  { SNAME }
```

- *Example 8.3.9:*

```
( P WHERE WEIGHT > WEIGHT ( 16.0 ) ) { P# }
UNION
( SP WHERE S# = S# ( 'S2' ) ) { P# }
```

8.4 Calculus *vs*. Algebra / 8.5 Computational Capabilities

These sections should be self-explanatory.

8.6 SQL Facilities

This section contains the principal discussion in the book of SQL retrieval operations (mainly SELECT). We include that discussion at this point in the chapter because SQL is (or, at least, is supposed to be) based on the tuple calculus specifically. *Note:* The important concept of **orthogonality** is also introduced in passing in this section.

The first paragraph of Section 8.6 includes the following remarks (slightly reworded here): "Some aspects of SQL are algebra-like, some are calculus-like, and some are neither ... We leave it as an exercise to figure out which aspects are based on the algebra, which on the calculus, and which on neither." Here's a partial answer to this exercise (we concentrate on SQL *table expressions* only, since such expressions are the only part of SQL for which the exercise really makes much sense):

- *Algebra:*

  UNION, INTERSECT, EXCEPT
  explicit JOIN

- *Calculus:*

  EXISTS
  range variables*

---

* SQL doesn't use the term "range variables"; rather, it talks about something it calls "correlation names"——but it never says exactly what such names *name!*

- *Neither of the above:*

  nested subqueries (?)
  GROUP BY (?), HAVING (?)
  nulls
  duplicate rows
  left-to-right column ordering

Note: **Nulls** are discussed in Chapter 19. **Duplicate rows** need to be discussed now, at least with respect to their effects on SQL queries. *Recommendation:* Always specify DISTINCT!——but be annoyed about it.

Explain the SQL WITH clause (which isn't quite the same as the **Tutorial D** WITH clause; loosely, the SQL WITH clause is based on *text substitution,* while the **Tutorial D** one is based on *subexpression evaluation*). By the way, note that the **Tutorial D** WITH clause can be used with the relational calculus as well as the relational algebra (of course).

You might want show algebraic and/or calculus formulations of some of the SQL examples in this section. Stress the point that the SQL formulations shown are very far from being the only ones possible.

The reader is asked to give some alternative join formulations of Example 8.6.11. Here are a couple of possibilities. Note the need for DISTINCT in both cases ...

```
SELECT DISTINCT S.SNAME
FROM   S, SP, P
WHERE  S.S# = SP.S#
AND    SP.P# = P.P#
AND    P.COLOR = COLOR ('Red') ;

SELECT DISTINCT S.SNAME
FROM   ( SELECT S#, SNAME FROM S ) AS POINTLESS1
       NATURAL JOIN
       SP
       NATURAL JOIN
       ( SELECT P#, COLOR FROM P ) AS POINTLESS2
WHERE  P.COLOR = COLOR ('Red') ;
```

I wouldn't discuss the point in class unless somebody asks about it, but you should at least be aware of the fact that (as mentioned in the notes on Chapter 7) SQL gets into a lot of trouble over union, intersection, and difference. One point that might be worth mentioning is that we can't always talk sensibly in SQL of "the" union (etc.) of a given pair of tables, because there might be more than one such.

8.7 Domain Calculus

You could skip this section even if you didn't skip the tuple calculus sections. Note, however, that the section isn't meant to stand alone——it does assume a familiarity with the basic ideas of the tuple calculus. Alternatively, you might just briefly cover QBE at an intuitive level and skip the domain calculus *per se*.

8.8 Query-By-Example

QBE is basically a syntactically sugared form of the domain calculus (more or less——it does also implicitly support the *tuple* calculus version of EXISTS). The section is more or less self-explanatory (as far as it goes, which deliberately isn't very far). The fact that QBE isn't relationally complete is probably worth mentioning.

Answers to Exercises

8.1 a. Not valid. b. Not valid. c. Valid. d. Valid. e. Not valid. f. Not valid. g. Not valid. ***Note:*** The reason e. isn't valid is that FORALL applied to an empty set yields TRUE, while EXISTS applied to an empty set yields FALSE. Thus, e.g, the fact that the statement "All purple parts weigh over 100 pounds" is true (i.e., is a true proposition) doesn't necessarily mean any purple parts actually exist.

We remark that the (valid!) equivalences and implications can be used as a basis for a set of calculus expression transformation rules, much like the algebraic expression transformation rules mentioned in Chapter 7 and discussed in detail in Chapter 18. An analogous remark applies to the answers to Exercises 8.2 and 8.3 as well.

8.2 a. Valid. b. Valid. c. Valid (this one was discussed in the body of the chapter). d. Valid (hence each of the quantifiers can be defined in terms of the other). e. Not valid. f. Valid. Observe that (as a. and b. show) a sequence of **like** quantifiers can be written in any order without changing the meaning, whereas (as e. shows) for **unlike** quantifiers the order is significant. By way of illustration of this latter point, let *x* and *y* range over the set of integers and let *p* be the WFF "*y* > *x*". Then it should be clear that the WFF

FORALL *x* EXISTS *y* (*y* > *x*)

("For all integers *x*, there exists a larger integer *y*") evaluates to TRUE, whereas the WFF

EXISTS *y* FORALL *x* (*y* > *x*)

("There exists an integer *x* that is larger than every integer *y*") evaluates to FALSE. Hence interchanging unlike quantifiers changes the meaning. In a calculus-based query language, therefore, interchanging unlike quantifiers in a WHERE clause will change the meaning of the query. See reference [8.3].

8.3 a. Valid. b. Valid.

Copyright (c) 2003 C. J. Date

8.4 If supplier S2 currently supplies no parts, the original query will return all supplier numbers currently appearing in S (including in particular S2, who presumably appears in S but not in SP). If we replace SX by SPX throughout, it will return all supplier numbers currently appearing in SP. The difference between the two formulations is thus as follows: The first means "Get supplier numbers for suppliers who supply at least all those parts supplied by supplier S2" (as required). The second means "Get supplier numbers for suppliers who ***supply at least one part and*** supply at least all those parts supplied by supplier S2."

8.5 a. Get part name and city for parts supplied to every project in Paris by every supplier in London in a quantity less than 500.
b. The result of this query is empty.

8.6 This exercise is very difficult!——especially when we take into account the fact that part weights aren't unique. (If they were, we could paraphrase the query as "Get all parts such that the count of heavier parts is less than three.") The exercise is so difficult, in fact, that we don't even attempt to give a pure calculus solution here. It illustrates very well the point that relational completeness is only a ***basic*** measure of expressive power, and probably not a sufficient one. (The next two exercises also illustrate this point.) See reference [7.5] for an extended discussion of queries of this type.

8.7 Let PSA, PSB, PSC, ..., PS*n* be range variables ranging over (the current value of) relvar PART_STRUCTURE, and suppose the given part is part P1. Then:

a. A calculus expression for the query "Get part numbers for all parts that are components, at the ***first*** level, of part P1" is:

```
PSA.MINOR_P# WHERE PSA.MAJOR_P# = P# ( 'P1' )
```

b. A calculus expression for the query "Get part numbers for all parts that are components, at the ***second*** level, of part P1" is:

```
PSB.MINOR_P# WHERE EXISTS PSA
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# )
```

c. A calculus expression for the query "Get part numbers for all parts that are components, at the ***third*** level, of part P1" is:

```
PSC.MINOR_P# WHERE EXISTS PSA EXISTS PSB
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# AND
                   PSC.MAJOR_P# = PSB.MINOR_P# )
```

And so on. A calculus expression for the query "Get part numbers for all parts that are components, at the *nth* level, of part P1" is:

```
PSn.MINOR_P# WHERE EXISTS PSA EXISTS PSB ... EXISTS PS(n-1)
                 ( PSA.MAJOR_P# = P# ( 'P1' ) AND
                   PSB.MAJOR_P# = PSA.MINOR_P# AND
                   PSC.MAJOR_P# = PSB.MINOR_P# AND
                      ......................    AND
                   PSn.MAJOR_P# = PS(n-1).MINOR_P# )
```

All of these result relations a., b., c., ... then need to be "unioned" together to construct the PART_BILL result.

The problem is, of course, that there's no way to write *n* such expressions if the value of *n* is unknown. In fact, the part explosion query is a classic illustration of a problem that can't be formulated by means of a single expression in a language that's only relationally complete——i.e., a language that's no more powerful than the original calculus (or algebra). We therefore need another extension to the original calculus (and algebra). The TCLOSE operator discussed briefly in Chapter 7 is part of the solution to this problem (but only part). Further details are beyond the scope of this book.

Note: Although this problem is usually referred to as "bill-of-materials" or "parts explosion," it's actually of much wider applicability than those names might suggest. In fact, the kind of relationship typified by the "parts contain parts" structure occurs in a very wide range of applications. Other examples include management hierarchies, family trees, authorization graphs, communication networks, software module invocation structures, transportation networks, etc., etc.

8.8 This query can't be expressed in either the calculus or the algebra. For example, to express it in the calculus, we would basically need to be able to say something like the following:

Does there exist a relation *r* such that there exists a tuple *t* in *r* such that *t*.S# = S#('S1')?

In other words, we would need to be able to quantify over *relations* instead of tuples, and we would therefore need a new kind of range variable, one that denoted relations instead of tuples. The query therefore can't be expressed in the relational calculus as currently defined.

Note, incidentally, that the query under discussion is a "yes/no" query (the desired answer is basically a truth value). You might be tempted to think, therefore, that the reason the

query can't be handled in the calculus or the algebra is that calculus and algebra expressions are relation-valued, not truth-valued. However, yes/no queries *can* be handled in the calculus and algebra if properly implemented! The crux of the matter is to recognize that yes and no (equivalently, TRUE and FALSE) are *representable as relations*. The relations in question are TABLE_DEE and TABLE_DUM, respectively.

8.9 In order to show that SQL is relationally complete, we have to show, first, (a) that there exist SQL expressions for each of the five primitive (algebraic) operators restrict, project, product, union, and difference, and then (b) that the operands to those SQL expressions can be arbitrary SQL expressions in turn.

We begin by observing that SQL effectively does support the relational algebra RENAME operator, thanks to the availability of the optional "AS *<column name>*" specification on items in the SELECT clause.* We can therefore ensure that all tables do have proper column names, and in particular that the operands to product, union, and difference satisfy the requirements of (our version of) the algebra with respect to column naming. Furthermore——provided those operand column-naming requirements are indeed satisfied——the SQL column name inheritance rules in fact coincide with those of the algebra as described (under the name *relation type inference*) in Chapter 7.

———————

* To state the matter a little more precisely: An SQL analog of the algebraic expression T RENAME A AS B is the (extremely inconvenient!) SQL expression SELECT A AS B, X, Y, ..., Z FROM T (where X, Y, ..., Z are all of the columns of T apart from A, and we choose to overlook the fact that the SQL expression results in a table with a left-to-right ordering to its columns).

———————

Here then are SQL expressions corresponding approximately to the five primitive operators:

Algebra	*SQL*
A WHERE *p*	SELECT * FROM *A* WHERE *p*
A { *X, Y, ..., Z* }	SELECT DISTINCT *X, Y, ..., Z* FROM *A*
A TIMES *B*	SELECT * FROM A, B
A UNION *B*	SELECT * FROM *A* UNION SELECT * FROM *B*

A MINUS *B* SELECT * FROM *A* EXCEPT SELECT * FROM *B*

Reference [4.20] shows that each of *A* and *B* in the SQL expressions shown above is in fact a *<table reference>*. It also shows that if we take any of the five SQL expressions shown and enclose it in parentheses, what results is in turn a *<table reference>*.* It follows that SQL is indeed relationally complete.

---

* We ignore the fact that SQL would in fact require such a *<table reference>* to include a pointless range variable definition.

Note: Actually there is a glitch in the foregoing——SQL fails to support *projection over no columns at all* (because it also fails to support empty SELECT clauses). As a consequence, it doesn't support TABLE_DEE or TABLE_DUM.

8.10 SQL supports EXTEND but not SUMMARIZE (at least, not very directly). Regarding EXTEND, the relational algebra expression

```
EXTEND A ADD exp AS Z
```

can be represented in SQL as

```
SELECT A.*, exp' AS Z
FROM   ( A ) AS A
```

The expression *exp'* in the SELECT clause is the SQL counterpart of the EXTEND operand *exp*. The parenthesized *A* in the FROM clause is a *<table reference>* of arbitrary complexity (corresponding to the EXTEND operand *A*); the other *A* in the FROM clause is a range variable name.

Regarding SUMMARIZE, the basic problem is that the relational algebra expression

```
SUMMARIZE A PER B ...
```

yields a result with cardinality equal to that of *B*, while the SQL "equivalent"

```
SELECT ...
FROM   A
GROUP  BY C ;
```

yields a result with cardinality equal to that of the projection of *A* over *C*.

8.11 SQL doesn't support relational comparisons directly. However, such operations can be simulated, albeit only in a very cumbersome manner. For example, the comparison

A = *B*

(where *A* and *B* are relvars) can be simulated by the SQL expression

```
NOT EXISTS ( SELECT * FROM A
             WHERE NOT EXISTS ( SELECT * FROM B
                                WHERE A-row = B-row ) )
AND
NOT EXISTS ( SELECT * FROM B
             WHERE NOT EXISTS ( SELECT * FROM A
                                WHERE B-row = A-row ) )
```

(where *A-row* and *B-row* are *<row value constructor>*s representing an entire row of *A* and an entire row of *B*, respectively).

8.12 Here are a few such formulations. Note that the following list isn't even close to being exhaustive [4.19]. Note too that this is a very simple query!

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = P#('P2') ) ;

SELECT DISTINCT T.SNAME
FROM ( S NATURAL JOIN SP ) AS T
WHERE  T.P# = P#('P2') ;

SELECT DISTINCT T.SNAME
FROM ( S JOIN SP ON S.S# = SP.P# AND SP.P# = P#('P2') ) AS T ;

SELECT DISTINCT T.SNAME
FROM ( S JOIN SP USING S# ) AS T
WHERE  T.P# = P#('P2') ;

SELECT DISTINCT S.SNAME
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
```

```
          WHERE  SP.S# = S.S#
          AND    SP.P# = P#('P2') ) ;

    SELECT DISTINCT S.SNAME
    FROM   S, SP
    WHERE  S.S# = SP.S#
    AND    SP.P# = P#('P2') ;

    SELECT DISTINCT S.SNAME
    FROM   S
    WHERE  0 <
         ( SELECT COUNT(*)
           FROM   SP
           WHERE  SP.S# = S.S#
           AND    SP.P# = P#('P2') ) ;

    SELECT DISTINCT S.SNAME
    FROM   S
    WHERE  P#('P2') IN
         ( SELECT SP.P#
           FROM   SP
           WHERE  SP.S# = S.S# ) ;

    SELECT S.SNAME
    FROM   S, SP
    WHERE  S.S# = SP.S#
    AND    SP.P# = P#('P2')
    GROUP  BY S.SNAME ;
```

Subsidiary question: What are the implications of the foregoing? *Answer:* The language is harder to document, teach, learn, remember, use, and implement efficiently, than it ought to be.

8.13 We've numbered the following solutions as 8.13.*n*, where 7.*n* is the number of the original exercise in Chapter 7. We assume that SX, SY, PX, PY, JX, JY, SPJX, SPJY (etc.) are range variables ranging over suppliers, parts, projects, and shipments, respectively; definitions of those range variables aren't shown.

8.13.13 JX

8.13.14 JX WHERE JX.CITY = 'London'

8.13.15 SPJX.S# WHERE SPJX.J# = J# ('J1')

8.13.16 SPJX WHERE SPJX.QTY ≥ QTY (300) AND
SPJX.QTY ≤ QTY (750)

8.13.17 { PX.COLOR, PX.CITY }

8.13.18 { SX.S#, PX.P#, JX.J# } WHERE SX.CITY = PX.CITY
AND PX.CITY = JX.CITY

8.13.19 { SX.S#, PX.P#, JX.J# } WHERE SX.CITY =/ PX.CITY
OR PX.CITY =/ JX.CITY
OR JX.CITY =/ SX.CITY

8.13.20 { SX.S#, PX.P#, JX.J# } WHERE SX.CITY =/ PX.CITY
AND PX.CITY =/ JX.CITY
AND JX.CITY =/ SX.CITY

8.13.21 SPJX.P# WHERE EXISTS SX (SX.S# = SPJX.S# AND
SX.CITY = 'London')

8.13.22 SPJX.P# WHERE EXISTS SX EXISTS JX
(SX.S# = SPJX.S# AND SX.CITY = 'London' AND
JX.J# = SPJX.J# AND JX.CITY = 'London')

8.13.23 { SX.CITY AS SCITY, JX.CITY AS JCITY }
WHERE EXISTS SPJX (SPJX.S# = SX.S# AND SPJX.J# = JX.J#)

8.13.24 SPJX.P# WHERE EXISTS SX EXISTS JX
(SX.CITY = JX.CITY AND
SPJX.S# = SX.S# AND
SPJX.J# = JX.J#)

8.13.25 SPJX.J# WHERE EXISTS SX EXISTS JX
(SX.CITY =/ JX.CITY AND
SPJX.S# = SX.S# AND
SPJX.J# = JX.J#)

8.13.26 { SPJX.P# AS XP#, SPJY.P# AS YP# }
WHERE SPJX.S# = SPJY.S# AND SPJX.P# < SPJY.P#

8.13.27 COUNT (SPJX.J# WHERE SPJX.S# = S# ('S1')) AS N

8.13.28 SUM (SPJX WHERE SPJX.S# = S# ('S1')
AND SPJX.P# = P# ('P1'), QTY) AS Q

Note: The following "solution" is ***not*** correct (why not?):

SUM (SPJX.QTY WHERE SPJX.S# = S# ('S1')
AND SPJX.P# = P# ('P1')) AS Q

Answer: Because duplicate QTY values will now be eliminated before the sum is computed.

8.13.29 { SPJX.P#, SPJX.J#,

Copyright (c) 2003 C. J. Date

```
          SUM ( SPJY WHERE SPJY.P# = SPJX.P#
                      AND   SPJY.J# = SPJX.J#, QTY ) AS Q }
```

8.13.30
```
SPJX.P# WHERE
AVG ( SPJY WHERE SPJY.P# = SPJX.P#
             AND   SPJY.J# = SPJX.J#, QTY ) > QTY ( 350 )
```

8.13.31
```
JX.JNAME WHERE EXISTS SPJX ( SPJX.J# = JX.J# AND
                             SPJX.S# = S# ( 'S1' ) )
```

8.13.32
```
PX.COLOR WHERE EXISTS SPJX ( SPJX.P# = PX.P# AND
                             SPJX.S# = S# ( 'S1' ) )
```

8.13.33
```
SPJX.P# WHERE EXISTS JX ( JX.CITY = 'London' AND
                          JX.J# = SPJX.J# )
```

8.13.34
```
SPJX.J# WHERE EXISTS SPJY ( SPJX.P# = SPJY.P# AND
                            SPJY.S# = S# ( 'S1' ) )
```

8.13.35
```
SPJX.S# WHERE EXISTS SPJY EXISTS SPJZ EXISTS PX
              ( SPJX.P# = SPJY.P# AND
                SPJY.S# = SPJZ.S# AND
                SPJZ.P# = PX.P# AND
                PX.COLOR = COLOR ( 'Red' ) )
```

8.13.36
```
SX.S# WHERE EXISTS SY ( SY.S# = S# ( 'S1' ) AND
                        SX.STATUS < SY.STATUS )
```

8.13.37
```
JX.J# WHERE FORALL JY ( JY.CITY ≥ JX.CITY )

Or:     JX.J# WHERE JX.CITY = MIN ( JY.CITY )
```

8.13.38
```
SPJX.J# WHERE SPJX.P# = P# ( 'P1' ) AND
          AVG ( SPJY WHERE SPJY.P# = P# ( 'P1' )
                      AND   SPJY.J# = SPJX.J#, QTY ) >
          MAX ( SPJZ.QTY WHERE SPJZ.J# = J# ( 'J1' ) )
```

8.13.39
```
SPJX.S# WHERE SPJX.P# = P# ( 'P1' )
          AND   SPJX.QTY >
                AVG ( SPJY
                      WHERE SPJY.P# = P# ( 'P1' )
                      AND   SPJY.J# = SPJX.J#, QTY )
```

8.13.40
```
JX.J# WHERE NOT EXISTS SPJX EXISTS SX EXISTS PX
            ( SX.CITY = 'London' AND
              PX.COLOR = COLOR ( 'Red' ) AND
              SPJX.S# = SX.S# AND
              SPJX.P# = PX.P# AND
              SPJX.J# = JX.J# )
```

Copyright (c) 2003 C. J. Date

8.13.41
```
JX.J# WHERE FORALL SPJY ( IF SPJY.J# = JX.J#
                            THEN SPJY.S# = S# ( 'S1' )
                            END IF )
```

8.13.42
```
PX.P# WHERE FORALL JX
        ( IF JX.CITY = 'London' THEN
          EXISTS SPJY ( SPJY.P# = PX.P# AND
                        SPJY.J# = JX.J# )
          END IF )
```

8.13.43
```
SX.S# WHERE EXISTS PX FORALL JX EXISTS SPJY
            ( SPJY.S# = SX.S# AND
              SPJY.P# = PX.P# AND
              SPJY.J# = JX.J# )
```

8.13.44
```
JX.J# WHERE FORALL SPJY ( IF SPJY.S# = S# ( 'S1' ) THEN
                            EXISTS SPJZ
                          ( SPJZ.J# = JX.J# AND
                            SPJZ.P# = SPJY.P# )
                            END IF )
```

8.13.45
```
RANGEVAR VX RANGES OVER
         ( SX.CITY ), ( PX.CITY ), ( JX.CITY ) ;

VX.CITY
```

8.13.46
```
SPJX.P# WHERE EXISTS SX ( SX.S# = SPJX.S# AND
                          SX.CITY = 'London' )
          OR    EXISTS JX ( JX.J# = SPJX.J# AND
                            JX.CITY = 'London' )
```

8.13.47
```
{ SX.S#, PX.P# }
WHERE NOT EXISTS SPJX ( SPJX.S# = SX.S# AND
                        SPJX.P# = PX.P# )
```

8.13.48
```
{ SX.S# AS XS#, SY.S# AS YS# }
WHERE FORALL PZ
  ( ( IF   EXISTS SPJX ( SPJX.S# = SX.S# AND
                         SPJX.P# = PZ.P# )
      THEN EXISTS SPJY ( SPJY.S# = SY.S# AND
                         SPJY.P# = PZ.P# )
      END IF )
    AND
     ( IF   EXISTS SPJY ( SPJY.S# = SY.S# AND
                          SPJY.P# = PZ.P# )
      THEN EXISTS SPJX ( SPJX.S# = SX.S# AND
                         SPJX.P# = PZ.P# )
      END IF ) )
```

8.13.49
```
{ SPJX.S#, SPJX.P#, { SPJY.J#, SPJY.QTY WHERE
```

Copyright (c) 2003 C. J. Date

```
                                SPJY.S# = SPJX.S# AND
                                SPJY.P# = SPJX.P# } AS JQ }
```

8.13.50 Let R denote the result of evaluating the expression shown in the previous solution. Then:

```
        RANGEVAR RX RANGES OVER R ,
        RANGEVAR RY RANGES OVER RX.JQ ;

        { RX.S#, RX.P#, RY.J#, RY.QTY }
```

We're extending the syntax and semantics of *<range var def>* slightly here. The idea is that the definition of RY depends on that of RX (note that the two definitions are separated by a comma, not a semicolon, and are thereby bundled into a single operation). See reference [3.3] for further discussion.

8.14 We've numbered the following solutions as 8.14.*n*, where 7.*n* is the number of the original exercise in Chapter 7.

8.14.13
```
        SELECT *
        FROM   J ;
```

Or simply:

```
        TABLE J ;
```

8.14.14
```
        SELECT J.*
        FROM   J
        WHERE  J.CITY = 'London' ;
```

8.14.15
```
        SELECT DISTINCT SPJ.S#
        FROM   SPJ
        WHERE  SPJ.J# = J#('J1') ;
```

8.14.16
```
        SELECT SPJ.*
        FROM   SPJ
        WHERE  SPJ.QTY >= QTY(300)
        AND    SPJ.QTY <= QTY(750) ;
```

8.14.17
```
        SELECT DISTINCT P.COLOR, P.CITY
        FROM   P ;
```

8.14.18
```
        SELECT S.S#, P.P#, J.J#
        FROM   S, P, J
        WHERE  S.CITY = P.CITY
        AND    P.CITY = J.CITY ;
```

8.14.19
```
        SELECT S.S#, P.P#, J.J#
        FROM   S, P, J
```

Copyright (c) 2003 C. J. Date

```
        WHERE  NOT ( S.CITY = P.CITY AND
                     P.CITY = J.CITY ) ;
```

8.14.20
```
        SELECT S.S#, P.P#, J.J#
        FROM   S, P, J
        WHERE  S.CITY <> P.CITY
        AND    P.CITY <> J.CITY
        AND    J.CITY <> P.CITY ;
```

8.14.21
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) = 'London' ;
```

8.14.22
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                WHERE  S.S# = SPJ.S# ) = 'London'
        AND    ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) = 'London' ;
```

8.14.23
```
        SELECT DISTINCT S.CITY AS SCITY, J.CITY AS JCITY
        FROM   S, J
        WHERE  EXISTS
             ( SELECT *
               FROM   SPJ
               WHERE  SPJ.S# = S.S#
               AND    SPJ.J# = J.J# ) ;
```

8.14.24
```
        SELECT DISTINCT SPJ.P#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) =
               ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) ;
```

8.14.25
```
        SELECT DISTINCT SPJ.J#
        FROM   SPJ
        WHERE  ( SELECT S.CITY
                 FROM   S
                 WHERE  S.S# = SPJ.S# ) <>
               ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) ;
```

Copyright (c) 2003 C. J. Date

8.14.26
```
SELECT DISTINCT SPJX.P# AS PA, SPJY.P# AS PB
FROM   SPJ AS SPJX, SPJ AS SPJY
WHERE  SPJX.S# = SPJY.S#
AND    SPJX.P# < SPJY.P# ;
```

8.14.27
```
SELECT COUNT ( DISTINCT SPJ.J# ) AS N
FROM   SPJ
WHERE  SPJ.S# = S#('S1') ;
```

8.14.28
```
SELECT SUM ( SPJ.QTY ) AS X
FROM   SPJ
WHERE  SPJ.S# = S#('S1')
AND    SPJ.P# = P#('P1') ;
```

8.14.29
```
SELECT SPJ.P#, SPJ.J#, SUM ( SPJ.QTY ) AS Y
FROM   SPJ
GROUP  BY SPJ.P#, SPJ.J# ;
```

8.14.30
```
SELECT DISTINCT SPJ.P#
FROM   SPJ
GROUP  BY SPJ.P#, SPJ.J#
HAVING AVG ( SPJ.QTY ) > QTY(350) ;
```

8.14.31
```
SELECT DISTINCT J.JNAME
FROM   J, SPJ
WHERE  J.J# = SPJ.J#
AND    SPJ.S# = S#('S1') ;
```

8.14.32
```
SELECT DISTINCT P.COLOR
FROM   P, SPJ
WHERE  P.P# = SPJ.P#
AND    SPJ.S# = S#('S1') ;
```

8.14.33
```
SELECT DISTINCT SPJ.P#
FROM   SPJ, J
WHERE  SPJ.J# = J.J#
AND    J.CITY = 'London' ;
```

8.14.34
```
SELECT DISTINCT SPJX.J#
FROM   SPJ AS SPJX, SPJ AS SPJY
WHERE  SPJX.P# = SPJY.P#
AND    SPJY.S# = S#('S1') ;
```

8.14.35
```
SELECT DISTINCT SPJX.S#
FROM   SPJ AS SPJX, SPJ AS SPJY, SPJ AS SPJZ
WHERE  SPJX.P# = SPJY.P#
AND    SPJY.S# = SPJZ.S#
AND  ( SELECT P.COLOR
       FROM   P
       WHERE  P.P# = SPJZ.P# ) = COLOR('Red') ;
```

Copyright (c) 2003 C. J. Date

8.14.36
```
SELECT S.S#
FROM   S
WHERE  S.STATUS < ( SELECT S.STATUS
                    FROM   S
                    WHERE  S.S# = S#('S1') ) ;
```

8.14.37
```
SELECT J.J#
FROM   J
WHERE  J.CITY = ( SELECT MIN ( J.CITY )
                  FROM   J ) ;
```

8.14.38
```
SELECT DISTINCT SPJX.J#
FROM   SPJ AS SPJX
WHERE  SPJX.P# = P#('P1')
AND  ( SELECT AVG ( SPJY.QTY )
       FROM   SPJ AS SPJY
       WHERE  SPJY.J# = SPJX.J#
       AND    SPJY.P# = P#('P1') ) >
     ( SELECT MAX ( SPJZ.QTY )
       FROM   SPJ AS SPJZ
       WHERE  SPJZ.J# = J#('J1') ) ;
```

8.14.39
```
SELECT DISTINCT SPJX.S#
FROM   SPJ AS SPJX
WHERE  SPJX.P# = P#('P1')
AND    SPJX.QTY > ( SELECT AVG ( SPJY.QTY )
                    FROM   SPJ AS SPJY
                    WHERE  SPJY.P# = P#('P1')
                    AND    SPJY.J# = SPJX.J# ) ;
```

8.14.40
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ, P, S
       WHERE  SPJ.J# = J.J#
       AND    SPJ.P# = P.P#
       AND    SPJ.S# = S.S#
       AND    P.COLOR = COLOR('Red')
       AND    S.CITY = 'London' ) ;
```

8.14.41
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ
       WHERE  SPJ.J# = J.J#
       AND    NOT ( SPJ.S# = S#('S1') ) ) ;
```

Copyright (c) 2003 C. J. Date

8.14.42
```
SELECT P.P#
FROM   P
WHERE  NOT EXISTS
     ( SELECT *
       FROM   J
       WHERE  J.CITY = 'London'
       AND    NOT EXISTS
            ( SELECT *
              FROM   SPJ
               WHERE  SPJ.P# = P.P#
              AND    SPJ.J# = J.J# ) ) ;
```

8.14.43
```
SELECT S.S#
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   J
              WHERE  NOT EXISTS
                   ( SELECT *
                  FROM   SPJ
                   WHERE  SPJ.S# = S.S#
                   AND    SPJ.P# = P.P#
                   AND    SPJ.J# = J.J# ) ) ) ;
```

8.14.44
```
SELECT J.J#
FROM   J
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SPJ AS SPJX
       WHERE  SPJX.S# = S#('S1')
       AND    NOT EXISTS
            ( SELECT *
              FROM   SPJ AS SPJY
             WHERE  SPJY.P# = SPJX.P#
              AND    SPJY.J# = J.J# ) ) ;
```

8.14.45
```
SELECT S.CITY FROM S
UNION
SELECT P.CITY FROM P
UNION
SELECT J.CITY FROM J ;
```

8.14.46
```
SELECT DISTINCT SPJ.P#
FROM   SPJ
WHERE  ( SELECT S.CITY
          FROM   S
         WHERE  S.S# = SPJ.S# ) = 'London'
```

Copyright (c) 2003 C. J. Date

```
        OR     ( SELECT J.CITY
                 FROM   J
                 WHERE  J.J# = SPJ.J# ) = 'London' ;
```

8.14.47
```
        SELECT S.S#, P.P#
        FROM   S, P
        EXCEPT
        SELECT SPJ.S#, SPJ.P#
        FROM   SPJ ;
```

8.14.48 *No answer provided.*

8.14.49-8.14.50 Can't be done (because SQL doesn't support relation-valued attributes).

8.15 We've numbered the following solutions as 8.15.*n*, where 7.*n* is the number of the original exercise in Chapter 7. We follow the same conventions as in Section 8.7 regarding the definition and naming of range variables.

8.15.13
```
        ( JX, NAMEX, CITYX )
        WHERE J ( J#:JX, JNAME:NAMEX, CITY:CITYX )
```

8.15.14
```
        ( JX, NAMEX, 'London' AS CITY )
        WHERE J ( J#:JX, JNAME:NAMEX, CITY:'London' )
```

8.15.15
```
        SX WHERE SPJ ( S#:SX, J#:J#('J1') )
```

8.15.16
```
        ( SX, PX, JX, QTYX )
        WHERE SPJ ( S#:SX, P#:PX, J#:JX, QTY:QTYX )
        AND   QTYX ≥ QTY ( 300 ) AND QTYX ≤ QTY ( 750 )
```

8.15.17
```
        ( COLORX, CITYX WHERE P ( COLOR:COLORX, CITY:CITYX ) )
```

8.15.18
```
        ( SX, PX, JX ) WHERE EXISTS CITYX
                             ( S ( S#:SX, CITY:CITYX ) AND
                               P ( P#:PX, CITY:CITYX ) AND
                               J ( J#:JX, CITY:CITYX ) )
```

8.15.19
```
        ( SX, PX, JX )
        WHERE EXISTS CITYX EXISTS CITYY EXISTS CITYZ
                        ( S ( S#:SX, CITY:CITYX ) AND
                          P ( P#:PX, CITY:CITYY ) AND
                          J ( J#:JX, CITY:CITYZ )
                          AND ( CITYX =/ CITYY OR
                                CITYY =/ CITYZ OR
                                CITYZ =/ CITYX ) )
```

8.15.20
```
        ( SX, PX, JX )
        WHERE EXISTS CITYX EXISTS CITYY EXISTS CITYZ
```

```
                        ( S ( S#:SX, CITY:CITYX ) AND
                          P ( P#:PX, CITY:CITYY ) AND
                          J ( J#:JX, CITY:CITYZ )
                          AND ( CITYX =/ CITYY AND
                                CITYY =/ CITYZ AND
                                CITYZ =/ CITYX ) )
```

8.15.21 `PX WHERE EXISTS SX ( SPJ ( P#:PX, S#:SX ) AND`
`S ( S#:SX, CITY:'London' ) )`

8.15.22
```
PX WHERE EXISTS SX EXISTS JX
               ( SPJ ( S#:SX, P#:PX, J#:JX )
                  AND S ( S#:SX, CITY:'London' )
                  AND J ( J#:JX, CITY:'London' )
```

8.15.23
```
( CITYX AS SCITY, CITYY AS JCITY )
               WHERE EXISTS SX EXISTS JY
                    ( S ( S#:SX, CITY:CITYX )
                      AND J ( J#:JY, CITY:CITYY )
                      AND SPJ ( S#:SX, J#:JY ) )
```

8.15.24
```
PX WHERE EXISTS SX EXISTS JX EXISTS CITYX
              ( S ( S#:SX, CITY:CITYX )
                AND J ( J#:JX, CITY:CITYX )
                AND SPJ ( S#:SX, P#:PX, J#:JX ) )
```

8.15.25
```
JY WHERE EXISTS SX EXISTS CITYX EXISTS CITYY
              ( SPJ ( S#:SX, J#:JY )
                AND S ( S#:SX, CITY:CITYX )
                AND J ( J#:JY, CITY:CITYY )
                AND CITYX =/ CITYY )
```

8.15.26
```
( PX AS XP#, PY AS YP# ) WHERE EXISTS SX
                                  ( SPJ ( S#:SX, P#:PX )
                                    AND SPJ ( S#:SX, P#:PY )
                                    AND PX < PY )
```

8.15.27-8.15.30 ***No answers provided*** (because Section 8.7 didn't discuss aggregate operators).

8.15.31
```
NAMEX WHERE EXISTS JX
                 ( J ( J#:JX, JNAME:NAMEX )
                   AND SPJ ( S#:S#('S1'), J#:JX ) )
```

8.15.32
```
COLORX WHERE EXISTS PX
                 ( P ( P#:PX, COLOR:COLORX ) AND
                   SPJ ( S#:S#('S1'), P#:PX ) )
```

8.15.33
```
PX WHERE EXISTS JX
              ( SPJ ( P#:PX, J#:JX ) AND
```

Copyright (c) 2003 C. J. Date

```
                J ( J#:JX, CITY:'London' ) )
```

8.15.34
```
JX WHERE EXISTS PX
              ( SPJ ( P#:PX, J#:JX ) AND
                SPJ ( P#:PX, S#:S#('S1') ) )
```

8.15.35
```
SX WHERE EXISTS PX EXISTS SY EXISTS PY
              ( SPJ ( S#:SX, P#:PX ) AND
                SPJ ( P#:PX, S#:SY ) AND
                SPJ ( S#:SY, P#:PY ) AND
               P ( P#:PY, COLOR:COLOR('Red') ) )
```

8.15.36
```
SX WHERE EXISTS STATUSX EXISTS STATUSY
              ( S ( S#:SX, STATUS:STATUSX ) AND
                S ( S#:S#('S1'), STATUS:STATUSY ) AND
                STATUSX < STATUSY )
```

8.15.37
```
JX WHERE EXISTS CITYX
              ( J ( J#:JX, CITY:CITYX ) AND
               FORALL CITYY ( IF J ( CITY:CITYY )
                              THEN CITYY ≥ CITYX
                              END IF )
```

8.15.38-8.15.39 *No answers provided* (because Section 8.7 didn't discuss aggregate operators).

8.15.40
```
JX WHERE J ( J#:JX ) AND
               NOT EXISTS SX EXISTS PX
                 ( SPJ ( S#:SX, P#:PX, J#:JX ) AND
                   S ( S#:SX, CITY:'London' ) AND
                   P ( P#:PX, COLOR:COLOR('Red') ) )
```

8.15.41
```
JX WHERE J ( J#:JX )
         AND   FORALL SX ( IF SPJ ( S#:SX, J#:JX )
                           THEN SX = S#('S1')
                           END IF )
```

8.15.42
```
PX WHERE P ( P#:PX )
         AND   FORALL JX ( IF J ( J#:JX, CITY:'London' )
                           THEN SPJ ( P#:PX, J#:JX )
                           END IF )
```

8.15.43
```
SX WHERE S ( S#:SX )
         AND   EXISTS PX FORALL JX
             ( SPJ ( S#:SX, P#:PX, J#:JX ) )
```

8.15.44
```
JX WHERE J ( J#:JX )
         AND   FORALL PX ( IF SPJ ( S#:S#('S1'), P#:PX )
                           THEN SPJ ( P#:PX, J#:JX )
                           END IF )
```

Copyright (c) 2003 C. J. Date

8.15.45
```
CITYX WHERE S ( CITY:CITYX )
            OR    P ( CITY:CITYX )
            OR    J ( CITY:CITYX )
```

8.15.46
```
PX WHERE EXISTS SX ( SPJ ( S#:SX, P#:PX ) AND
                           S ( S#:SX, CITY:'London' ) )
          OR    EXISTS JX ( SPJ ( J#:JX, P#:PX ) AND
                           J ( J#:JX, CITY:'London' ) )
```

8.15.47
```
( SX, PX ) WHERE S ( S#:SX ) AND P ( P#:PX )
                       AND NOT SPJ ( S#:SX, P#:PX )
```

8.15.48
```
( SX AS XS#, SY AS YS# )
      WHERE S ( S#:SX ) AND S ( S#:SY ) AND FORALL PZ
        ( ( IF SPJ ( S#:SX, P#:PZ ) THEN SPJ ( S#:SY, P#:PZ )
            END IF )
          AND
          ( IF SPJ ( S#:SY, P#:PZ ) THEN SPJ ( S#:SX, P#:PZ )
            END IF ) )
```

8.15.49-8.15.50 *No answers provided* (because Section 8.7 didn't discuss grouping and ungrouping).

8.16 *No answers provided* (because some of the queries are trivial; others can't be done using only the subset of QBE sketched in the chapter; and others can't be done in QBE at all, since QBE isn't relationally complete).

***** End of Chapter 8 *****

Copyright (c) 2003 C. J. Date

Chapter 9

I n t e g r i t y

Principal Sections

- A closer look
- Predicates and propositions
- Relvar predicates and DB predicates
- Checking the constraints
- Internal *vs.* external predicates
- Correctness *vs.* consistency
- Integrity and views
- A constraint classification scheme
- Keys
- Triggers (a digression)
- SQL facilities

General Remarks

This chapter has been rewritten from beginning to end in the eighth edition (even the division into sections has changed drastically). It's based in part on reference [9.16]. ***It's also one of the most important chapters in the entire book.***

What's the most important thing about a database? Surely it's to make sure——insofar as possible——that the data it contains is **correct!** This simple observation is the justification for my constant claims to the effect that integrity is crucial and fundamental, and that (as stated in Chapter 1) a database isn't really just a collection of "data"——rather, it's a collection of "true facts" (i.e., true propositions).

This heavy emphasis on integrity (or **semantics,** if you prefer) is yet another feature that sets this book apart from its competitors. In fact, I did a quick survey of a whole shelfload of database textbooks——37 in all (!)——and found that:

- *Only one* had an entire chapter devoted to the topic, and that chapter was structured and balanced very strangely (the sections were entitled "Domain Constraints,"* "Referential Integrity," "Assertions," "Triggers," and "Functional Dependencies"; this seems to me a little bit like having a chapter in a biology text with sections entitled "Flightless

Copyright (c) 2003 C. J. Date

Birds, Birds of Prey, Birds, How Flying Birds Fly, and Sparrows").

---

* Corresponding to a mixture of *type* and *attribute* constraints, in terms of the classification scheme presented in the book.

Note: At first glance it looked as if there were three others that had a whole chapter on the subject too, but closer examination revealed that one was using the term to refer to normalization issues solely, while the other two were using it to refer, not to integrity in its usual sense at all, but rather to locking and concurrency control issues. *Caveat lector!*

- Most of the books didn't even mention integrity in a chapter title at all, and those that did tended to bundle it with other topics in what seemed a very haphazard fashion ("Integrity, Views, Security, and Catalogs"——?!?——is a typical example).

- The coverage in the rest was limited to one section each, typically in a chapter on SQL. (In fact, the coverage in *all* of the books tended to be SQL-specific, and SQL is *not* a good basis for explaining integrity!——see Section 9.12.)

- I couldn't find a good *explanation* or *definition* of the concept, let alone the kind of emphasis I think the concept deserves, in any of the books at all.

The **classification scheme** for integrity constraints——into type, attribute, relvar, and database constraints——described in this chapter is based on work done by myself with David McGoveran. It's more logical and more systematic than some other schemes described in the literature. Note in particular that relvar and database constraints are crucial to the view updating mechanism described in Chapter 10, and type constraints are crucial to the type inheritance mechanism described in Chapter 20. As for attribute constraints, they're essentially trivial, for reasons explained in Section 9.9, subsection "Attribute Constraints."

It's worth stressing that integrity is not "just keys." Integrity has to do with *semantics* or *"business rules,"* and "business rules" can be arbitrarily complex; key constraints are just an important special case.

All of the "preliminary points" in Section 9.1 need to be covered.

We use the calculus, not the algebra, as a basis for our examples "for reasons that should become clear as we proceed" (as the text puts it).* The choice is arbitrary, of course, but in this context, at least, it does seem as if the fact that "the calculus is closer to natural language" argues in its favor. *Note:* Section 9.1 and Exercise 9.2 both suggest that the reader try converting certain calculus-based constraints into algebraic form. *No answer provided.*

---

* This shouldn't be a problem even if you skipped Chapter 8. The constraint formulations aren't hard to follow, even without a deep knowledge of the calculus——though an understanding of the quantifiers will surely help.

9.2 A Closer Look

Constraints apply to *variables,* not values. We're interested in *relation* variables specifically. Attribute declared types represent an *a priori* constraint on the relvar (explain), but there's much more to it!

Explain the general form of a constraint:

- ***IF** certain tuples appear in certain relvars, **THEN** those tuples satisfy a certain condition.*

Common special cases:

- ***IF** certain tuples appear in a certain relvar, **THEN** those tuples satisfy a certain condition.*
- ***IF** a certain tuple appears in a certain relvar, **THEN** that tuple satisfies a certain condition.*

Explain the terms *logical implication, antecedent, consequent.* Go through the six examples (each illustrates one new point). Note that (candidate) KEY and FOREIGN KEY constraints can be expressed, albeit longwindedly, using this general constraint language. Use either **Tutorial D** or pure calculus——not both——for examples in class (**Tutorial D** is probably the better choice).

9.3 Predicates and Propositions

People are frequently confused over the message of this section, which is SIMPLE but IMPORTANT:

- A constraint as formally stated is a *predicate*.
- When that constraint is checked, arguments are substituted for the parameters and the predicate is thereby reduced to a *proposition*——and that proposition is then required to evaluate to TRUE.

The parameters in question stand for relvars. Don't get into it yet, but the predicate in question is an **internal** predicate specifically (we'll get into internal *vs.* external predicates——another important logical difference!——in Section 9.6).

9.4 Relvar and DB Predicates

The material of this section is of fundamental importance. Unfortunately, however, it isn't very widely supported in practice, nor even much understood, even though in principle it's quite straightforward. It's also not much discussed in the literature!

Explain *relvar predicates* (and note slight shift in meaning since the seventh edition). **The Golden Rule** (first version):

> *No update operation must ever assign to any relvar a value that causes its relvar predicate to evaluate to FALSE.*

The text doesn't say this explicitly, but the rule applies to *all* relvars, derived as well as base (in particular, it applies to views——forward pointer to Chapter 10).

Explain *database predicates* and **The Golden Rule** (second and final version):

> *No update operation must ever assign to any database a value that causes its database predicate to evaluate to FALSE.*

Predicates are *the criterion for the acceptability of updates.*

9.5 Checking the Constraints

Implementation issue (not very interesting, it's essentially just a matter of optimization): Check the constraints *before* doing the update, if possible. Model issue (VERY important, and a violation of "conventional wisdom"): **All constraint checking is immediate!**——constraints are satisfied *at statement boundaries*——no deferred (COMMIT-time) checking at all. Explain that this position (a) is unconventional and (b) will be justified later (in Chapter 16).

Note: In the seventh edition, I said that *database* constraints (see Section 9.9) should be checked at COMMIT time, not immediately, and went on to say:

> "It would be possible to insist that database constraints be checked immediately ... Whether it would be desirable is another matter, however, one that's still under investigation. My own current feeling——subject to possible revision!——is that it would *not* be desirable."

Well, I did "revise" my feelings on the matter. See Chapter 16.

9.6 Internal *vs*. External Predicates

Another IMPORTANT section ... You might have noticed that in this chapter so far we've been using the term *predicate* in a sense slightly different from that in which we used it in earlier chapters (Chapters 3 and 6 in particular). Now we need to clarify ... Be *very clear* on the difference between the formal, internal, system-understood predicate for a given relvar (the **relvar predicate** for that relvar) and the corresponding *in*formal, *ex*ternal, *user*-understood predicate. Ditto for databases, *mutatis mutandis*. Note that from this point forward the unqualified term *predicate* is used in the book to mean an internal predicate specifically (barring explicit statements to the contrary).

The *Closed World Assumption* applies to external predicates, not internal ones. (A tuple might satisfy some relvar predicate——an interval predicate, that is——and yet validly not appear in the corresponding relvar.) The next section elaborates.

9.7 Correctness *vs*. Consistency

Another HUGELY important section (and a logical difference with a vengeance) ... *The system cannot enforce truth, only consistency.* As the chapter says, **correct** implies **consistent** (but not the other way around), and **inconsistent** implies **incorrect** (but not the other way around)——where by *correct* we mean the database is correct if

and only if it fully reflects the true state of affairs in the real world.

9.8 Integrity and Views

Self-explanatory, but once again important——and a trifle unorthodox (most people think integrity applies to base relvars only). The only slightly tricky point is in Example 2 (projection involves the introduction of an EXISTS corresponding to the attribute that has been projected away). We'll be appealing to these ideas in the next chapter in particular.

9.9 A Constraint Classification Scheme

There have been many attempts (most of them not very successful) to come up with a sensible classification scheme for integrity constraints. I've made several such attempts myself!——see among other things my ***Relational Database Writings*** series, especially the 1991-1994 and 1994-1997 volumes (Addison-Wesley, 1995 and 1998, respectively); see also the two editions, coauthored with Hugh Darwen, of the ***Third Manifesto*** book. Other writers who have also tried to come up with classification schemes include:

- Ted Codd (in reference [6.2])
- Mike Stonebraker (1975 ACM SIGMOD Conference Proceedings)
- Jeff Ullman and Jennifer Widom (***A First Course in Database Systems,*** Prentice Hall, 1997)
- Ralph Kimball (***Intelligent Enterprise 3,*** Nos. 11 and 12, August 1st and 18th, 2000, respectively)
- Ron Ross (***The Business Rule Book: Classifying, Defining, and Modeling Rules,*** 2nd edition, Business Rule Solutions LLC, 1997)

(Not to mention the SQL standard——see Section 9.12.) The scheme presented in this section has a feeling of "rightness" about it, however, in that the structure of the classification mirrors the structure of the data itself. Databases are made out of relvars; relvars are made out of attributes; attributes are made out of types. So:

- A database constraint is a constraint on the values a given database is permitted to assume.

- A relvar constraint is a constraint on the values a given relvar is permitted to assume.

- An attribute constraint is a constraint on the values a given attribute is permitted to assume.

- A type constraint is, precisely, a definition of the set of values that constitute a given type.

Type constraints: Already discussed in Chapter 5——but we didn't explain in that chapter that such constraints are, at least conceptually, always checked ("immediately") during the execution of some ***selector invocation***. A type constraint is, precisely, a specification of the values that make up the type in question. The declared type of possrep components is an ***a priori*** constraint, but further constraints are possible. No relvar can ever acquire a value for any attribute in any tuple that isn't of the appropriate type (in a system that supports type constraints properly, of course, which unfortunately excludes all of today's SQL systems ... see Section 9.12!).

Attribute constraints: Self-explanatory. Note that if the system enforces type constraints properly, attribute constraints can never be violated.

Relvar constraints: Can be arbitrarily complex, so long as they explicitly refer to exactly one relvar. Candidate key constraints, discussed in detail in the next section, are an important special case. The relvar in question isn't necessarily a base relvar (see Chapter 10).

DB constraints: Can be arbitrarily complex, so long as it explicitly refers to at least two relvars. Foreign key constraints, discussed in detail in the next section 9.8, are an important special case (unless the referenced and referencing relvar happen to be one and the same, in which case the foreign key constraint is a ***relvar*** constraint instead). The database in question isn't necessarily the "real" database (see Chapter 10).

Relvar and database constraints are what we've been concentrating on in this chapter so far. The difference between them isn't——and in fact ***can't*** be, thanks to ***The Principle of Interchangeability of Base and Derived Relvars,*** which we'll be discussing in the next chapter——very important from a theoretical point of view, though it might be useful from a pragmatic one.

State *vs*. transition constraints: Self-explanatory——but note that transition constraints aren't much supported (if at all) in practice, despite the fact that they're very common in the real world. They're certainly not supported in SQL. (***Note:*** They can

be enforced by means of SQL triggers, of course, but I don't regard such procedural enforcement as proper "support." The whole point of all this stuff is that we want *declarative* support for everything, insofar as declarative support is possible. Further discussion to follow in Section 9.11.)

9.10 Keys

Keys are a *logical* notion, not a physical one. They don't apply just to base relvars! The book espouses at least two slightly heretical positions with respect to keys:

- Relvars must have at least one *candidate* key but not necessarily a primary key.

- *Foreign* keys must reference a candidate key but not necessarily a primary key.

Detailed arguments in defense of these positions can be found in reference [9.14].

Stress the fact that keys are *sets* of attributes and key values are therefore *sets* of attribute values; in fact, a key value is a (sub)tuple, and we're appealing to the notion of tuple equality once again. Syntax: Commalist of attribute names enclosed in *braces*.

Regarding **candidate** keys: Note that *irreducibility* is referred to as *minimality* in much of the literature. Mention **superkeys**.

Regarding **foreign** keys: Note the requirement——analogous to the requirement for, e.g., join——that each attribute of a given foreign key must have the same *name* (as well as the same type, of course) as the corresponding attribute of the matching candidate key; formally speaking, in fact, they're the *same* attribute.

The discussion of foreign keys includes this example of a "self-referencing" relvar:

```
VAR EMP BASE RELATION
          { EMP# EMP#, ..., MGR_EMP# EMP#, ... }
    PRIMARY KEY { EMP# }
    FOREIGN KEY { RENAME MGR_EMP# AS EMP# } REFERENCES EMP ;
```

It goes on to ask the reader to invent some sample data for this example. One possible answer to this exercise is as follows:

EMP

EMP#	...	MGR_EMP#	...
E1	...	E1	...
E2	...	E1	...
E3	...	E2	...
E4	...	E2	...

Note in passing that the manager for employee E1 is shown as employee E1, ***not*** as some kind of "null"!——i.e., E1 is his or her own manager.

Actually, a design in which the employee-to-manager relationship is split out into a separate relvar would probably be preferable, as here:

EMP

EMP#	...	...
E1	...	...
E2	...	...
E3	...	...
E4	...	...

EMM

EMP#	MGR_EMP#
E2	E1
E3	E2
E4	E2

Observe that EMM includes no tuple with EMP# = employee number E1, and the referential constraint is now a database constraint (it spans two relvars), not a relvar constraint. See reference [19.19] for further discussion of this kind of design.

Regarding the **referential integrity** rule: Note (a) the remark to the effect that the rule can be regarded as a "metaconstraint"; (b) the fact that discussion of its companion "metaconstraint," the ***entity*** integrity rule, is deferred to Chapter 19 (because it has to do with nulls).

Regarding **referential actions:** Use these ideas as a springboard for a brief discussion of ***triggered procedures*** (see the next section), but stress the fact that referential actions are specified ***declaratively,*** not procedurally, and declarative solutions are always preferable (because declarative means the system does the work, while procedural means the user does).

9.11 Triggers (a digression)

This section could be skipped; it's included here mainly for completeness (also because there's no other obvious place to put

it). Everything else in the chapter is important; triggers are more of a pragmatic issue. In fact, I think they represent an abdication of responsibility on the part of the vendor: "We don't know how to solve this problem, so we'll punt and pass it back to the user" (who now has to write a bunch of procedural code). If you do cover them, stress the point that triggers are much more useful for other purposes than they are for constraint checking, for which they're *not* the recommended solution.

9.12 SQL Facilities

Mostly self-explanatory ... but note that "self-explanatory" is not the same thing as making *sense*.

Answers to Exercises

9.1

1. INSERT on S, UPDATE on S.STATUS

2. INSERT on S, UPDATE on S.STATUS, UPDATE on S.CITY

3. INSERT on P, DELETE on P, UPDATE on P.COLOR

4. INSERT on S, UPDATE on S.S#

5. INSERT on SP, DELETE on S, UPDATE on SP.S#, UPDATE on S.S#

6. INSERT on SP, UPDATE on S.S#, UPDATE on S.STATUS, UPDATE on SP.S#, UPDATE on SP.QTY

9.2

1. CONSTRAINT SC1
   IS_EMPTY (S WHERE STATUS < 1 OR STATUS > 100) ;

2. CONSTRAINT SC2
   IS_EMPTY (S WHERE CITY = 'London' AND STATUS =/ 20) ;

3. CONSTRAINT PC3 IS_EMPTY (P) OR
   NOT (IS_EMPTY (P WHERE COLOR = COLOR ('Blue'))) ;

4. CONSTRAINT SC4 COUNT (S) = COUNT (S { S# }) ;

5. CONSTRAINT SSP5 SP { S# } ⊆ S { S# } ;

```
6. CONSTRAINT SSP6 IS_EMPTY ( ( S JOIN SP )
                               WHERE STATUS < 20
                               AND   QTY > QTY ( 500 ) ) ;
```

9.3

```
a. TYPE CITY
   POSSREP { C CHAR CONSTRAINT C = 'London'
                               OR C = 'Paris'
                               OR C = 'Rome'
                               OR C = 'Athens'
                               OR C = 'Oslo'
                               OR C = 'Stockholm'
                               OR C = 'Madrid'
                               OR C = 'Amsterdam' } ;
```

An obvious shorthand would be:

```
a. TYPE CITY
   POSSREP { C CHAR CONSTRAINT C IN { 'London', 'Paris',
                                      'Rome', 'Athens',
                                      'Oslo', 'Stockholm',
                                      'Madrid', 'Amsterdam' } ;
```

Note: A better solution might be to keep the legal city names in a relvar and to use foreign keys to ensure that no other relvar ever includes a city name that isn't one of the legal ones (this approach is likely to be more forgiving if a new city becomes legal). Such a solution would thus replace the foregoing type constraint (and corresponding attribute constraints) by a set of database constraints.

```
b. TYPE S# POSSREP { C CHAR CONSTRAINT
        LENGTH ( C ) ≥ 2 AND LENGTH ( C ) ≤ 5 AND
        SUBSTR ( C, 1, 1 ) = 'S' AND
        CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ≥ 0 AND
        CAST_AS_INTEGER ( SUBSTR ( C, 2 ) ≤ 9999 } ;
```

We assume here that the operators LENGTH, SUBSTR, and CAST_AS_INTEGER are available and have the obvious semantics.

```
c. CONSTRAINT C FORALL PX ( IF PX.COLOR = COLOR ( 'Red' )
                            THEN PX.WEIGHT < WEIGHT ( 50.0 )
                             END IF ) ;
```

Here and throughout the rest of these answers we follow our usual conventions regarding the definition and naming of range variables.

d. CONSTRAINT D
   FORALL JX FORALL JY (IF JX.J# =/ JY.J#
                         THEN JX.CITY =/ JY.CITY END IF) ;

e. CONSTRAINT E COUNT (SX WHERE SX.CITY = 'Athens') ≤ 1 ;

f. CONSTRAINT F
   FORALL SPJX (SPJX.QTY ≤ 2 * AVG (SPJY, QTY)) ;

g. CONSTRAINT G
   FORALL SX FORALL SY (IF SX.STATUS = MAX (S, STATUS) AND
                            SY.STATUS = MIN (S, STATUS)
                         THEN SX.CITY =/ SY.CITY END IF) ;

   Actually, the terms "highest status supplier" and "lowest status supplier" aren't well-defined, since status values aren't unique. We've interpreted the requirement to be that if S*x* and S*y* are ***any*** suppliers with "highest status" and "lowest status," respectively, then S*x* and S*y* mustn't be colocated. Note that the constraint will necessarily be violated if the "highest" and "lowest" status are equal!——in particular, it'll be violated if there's just one supplier. We could fix this problem by inserting AND SX.STATUS =/ SY.STATUS immediately before the THEN.

h. CONSTRAINT H FORALL JX EXISTS SX (SX.CITY = JX.CITY) ;

i. CONSTRAINT I FORALL JX EXISTS SX EXISTS SPJX
            (SX.CITY = JX.CITY AND
              SX.S# = SPJX.S# AND SPJX.J# = JX.J#) ;

j. CONSTRAINT J EXISTS PX (PX.COLOR = COLOR ('Red')) ;

   This constraint will be violated if there are no parts at all. A better formulation might be:

   CONSTRAINT J NOT EXISTS PX (TRUE) OR
              EXISTS PX (PX.COLOR = COLOR ('Red')) ;

k. CONSTRAINT K FORALL SX (AVG (SY, STATUS) > 19) ;

   The initial "FORALL SX" here is to avoid the error that would otherwise occur if the system tried to check the constraint when there were no suppliers at all.

l. CONSTRAINT L
   FORALL SX (IF SX.CITY = 'London' THEN
               EXISTS SPJX (SPJX.S# = SX.S# AND
                             SPJX.P# = P# ('P2') END IF) ;

Copyright (c) 2003 C. J. Date

```
m. CONSTRAINT M NOT EXISTS PX ( PX.COLOR = COLOR ( 'Red' ) ) OR
                EXISTS PX ( PX.COLOR = COLOR ( 'Red' ) AND
                            PX.WEIGHT < WEIGHT ( 50.0 ) ) ;

n. CONSTRAINT N
      COUNT ( SPJX.P# WHERE
              EXISTS SX ( SX.S# = SPJX.S# AND
                          SX.CITY = 'London' ) ) >
      COUNT ( SPJY.P# WHERE
              EXISTS SY ( SY.S# = SPJY.S# AND
                          SY.CITY = 'Paris'  ) ) ;

o. CONSTRAINT O
      SUM ( SPJX WHERE
            EXISTS SX ( SX.S# = SPJX.S# AND
                        SX.CITY = 'London' ), QTY ) >
      SUM ( SPJY WHERE
            EXISTS SY ( SY.S# = SPJY.S# AND
                        SY.CITY = 'Paris'  ), QTY ) ;

p. CONSTRAINT P
      FORALL SPJX' FORALL SPJX ( SPJX'.S# =/ SPJX.S# OR
                                 SPJX'.P# =/ SPJX.P# OR
                                 SPJX'.J# =/ SPJX.J# OR
                                 0.5 * SPJX'.QTY ≤ SPJX.QTY ) ;

q. CONSTRAINT Q
      FORALL SX' FORALL SX ( SX'.S# =/ SX.S# OR
                             ( IF SX'.CITY = 'Athens' THEN
                                  SX. CITY = 'Athens' OR
                                  SX. CITY = 'London' OR
                                  SX. CITY = 'Paris' END IF ) OR
                             ( IF SX'.CITY = 'London' THEN
                                  SX. CITY = 'London' OR
                                  SX. CITY = 'Paris' END IF ) ) ;
```

9.4 Constraints A and B are type constraints, of course. Of the others, constraints C, D, E, F, G, J, K, M, P, and Q are relvar constraints, the rest are database constraints. The operations that might cause the constraints to be violated are as follows:

a. CITY selector invocation

b. S# selector invocation

c. INSERT on P, UPDATE on P.WEIGHT

d. INSERT on J, UPDATE on J.CITY

e. INSERT on S, UPDATE on S.CITY

f. INSERT on SPJ, DELETE on SPJ, UPDATE on SPJ.QTY

g. INSERT on S, DELETE on S, UPDATE on S.STATUS, UPDATE on S.CITY

h. INSERT on J, DELETE on S, UPDATE on S.CITY, UPDATE on J.CITY

i. INSERT on J, DELETE on S, DELETE on SPJ, UPDATE on S.CITY, UPDATE on J.CITY, UPDATE on SPJ.S#, UPDATE on SPJ.J#

j. INSERT on P, DELETE on P, UPDATE on P.COLOR

k. INSERT on S, DELETE on S, UPDATE on S.STATUS

l. INSERT on S, DELETE on SPJ, UPDATE on S.S#, UPDATE on S.CITY, UPDATE on SPJ.S#, UPDATE on SPJ.P#

m. INSERT on P, DELETE on P, UPDATE on P.COLOR, UPDATE on P.WEIGHT

n. INSERT on S, INSERT on SPJ, DELETE on S, DELETE on SPJ, UPDATE on S.S#, UPDATE on S.CITY, UPDATE on SPJ.S#, UPDATE on SPJ.P#

o. INSERT on S, INSERT on SPJ, DELETE on S, DELETE on SPJ, UPDATE on S.S#, UPDATE on S.CITY, UPDATE on SPJ.S#, UPDATE on SPJ.QTY

p. UPDATE on SPJ.QTY

q. UPDATE on S.CITY

9.5

a. Accepted

b. Rejected (candidate key uniqueness violation)

c. Rejected (violates RESTRICT specification)

d. Accepted (supplier S3 and all shipments for supplier S3 are deleted)

e. Rejected (violates RESTRICT specification)

f. Accepted (project J4 and all shipments for project J4 are deleted)

g. Accepted

h. Rejected (candidate key uniqueness violation)

i. Rejected (referential integrity violation)

j. Accepted

k. Rejected (referential integrity violation)

l. Rejected (referential integrity violation——the default project number ***jjj*** does not exist in relvar J)

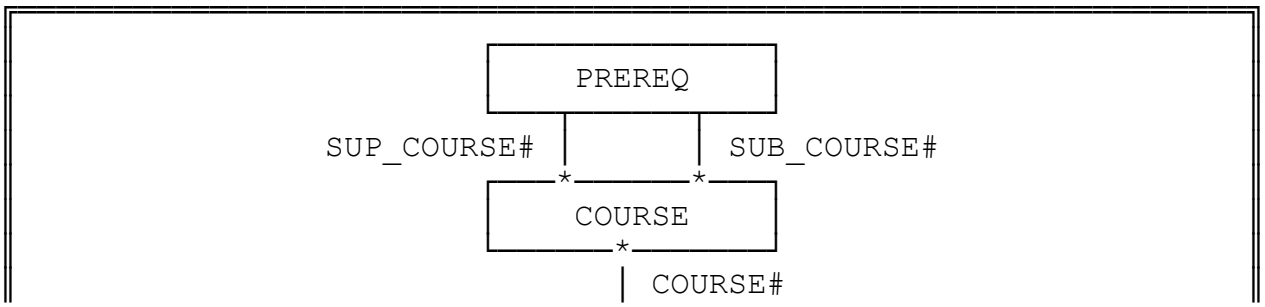
9.6 There's no explicit foreign key INSERT rule, because INSERTs on the referencing relvar——also UPDATEs on the foreign key in the referencing relvar——are governed by the basic referential integrity rule itself (i.e., the requirement that there be no unmatched foreign key values). In other words, taking suppliers and parts as a concrete example:

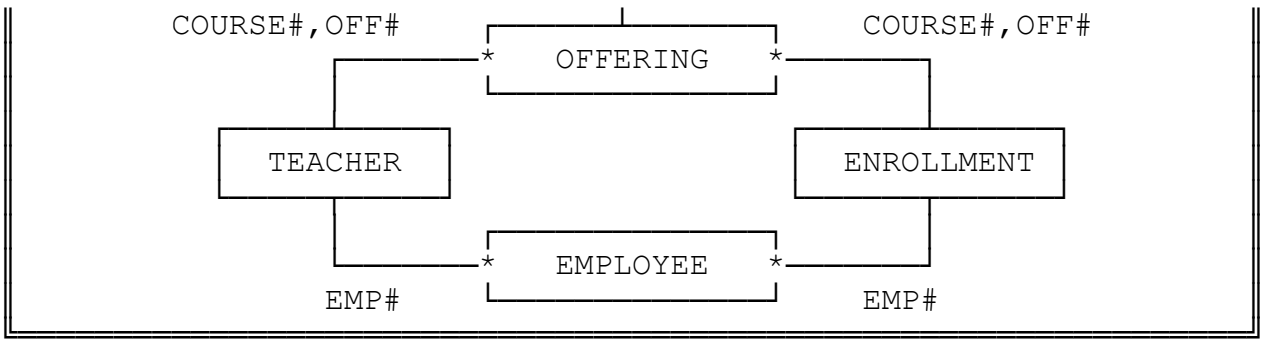
- An attempt to INSERT a shipment (SP) tuple will succeed only if (a) the supplier number in that tuple exists as a supplier number in S, ***and*** (b) the part number in that tuple exists as a part number in P.

- An attempt to UPDATE a shipment (SP) tuple will succeed only if (a) the supplier number in the updated tuple exists as a supplier number in S, ***and*** (b) the part number in the updated tuple exists as a part number in P.

Note carefully also that the foregoing remarks apply to the ***referencing*** relvar, whereas the (explicit) DELETE and UPDATE rules apply to the ***referenced*** relvar. Thus, to talk about an "INSERT rule," as if such a rule were somehow similar to the existing DELETE and UPDATE rules, is really a rather confusing thing to do. This fact provides additional justification for not including any explicit "INSERT rule" support in the concrete syntax.

9.7 The referential diagram is shown in the figure below. A possible database definition follows. For simplicity, we haven't bothered to define any type constraints——except inasmuch as the POSSREP specification on a given type definition serves as an ***a priori*** constraint on the type, of course.






```
TYPE COURSE# POSSREP { CHAR } ;
TYPE TITLE   POSSREP { CHAR } ;
TYPE OFF#    POSSREP { CHAR } ;
TYPE OFFDATE POSSREP { DATE } ;
TYPE CITY    POSSREP { CHAR } ;
TYPE EMP#    POSSREP { CHAR } ;
TYPE NAME    POSSREP { NAME } ;
TYPE JOB     POSSREP { CHAR } ;
TYPE GRADE   POSSREP { CHAR } ;

VAR COURSE BASE RELATION
     { COURSE# COURSE#,
       TITLE   TITLE }
     PRIMARY KEY { COURSE# } ;

VAR PREREQ BASE RELATION
     { SUP_COURSE# COURSE#,
       SUB_COURSE# COURSE# }
     KEY { SUP_COURSE#, SUB_COURSE# }
     FOREIGN KEY { RENAME SUP_COURSE# AS COURSE# }
                                      REFERENCES COURSE
                                      ON DELETE CASCADE
                                      ON UPDATE CASCADE
     FOREIGN KEY { RENAME SUB_COURSE# AS COURSE# }
                                      REFERENCES COURSE
                                      ON DELETE CASCADE
                                      ON UPDATE CASCADE ;

VAR OFFERING BASE RELATION
     { COURSE#  COURSE#,
       OFF#     OFF#,
       OFFDATE  OFFDATE,
       LOCATION CITY }
     KEY { COURSE#, OFF# }
     FOREIGN KEY { COURSE# } REFERENCES COURSE
                             ON DELETE CASCADE
                             ON UPDATE CASCADE ;

VAR EMPLOYEE BASE RELATION
     { EMP#  EMP#,
```

```
        ENAME NAME,
        JOB   JOB }
      KEY { EMP# } ;

VAR TEACHER BASE RELATION
      { COURSE# COURSE#,
        OFF#    OFF#,
        EMP#    EMP# }
      KEY { COURSE#, OFF#, EMP# }
      FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
                                    ON DELETE CASCADE
                                    ON UPDATE CASCADE
      FOREIGN KEY { EMP# } REFERENCES EMPLOYEE
                           ON DELETE CASCADE
                           ON UPDATE CASCADE ;

VAR ENROLLMENT BASE RELATION ENROLLMENT
      { COURSE# COURSE#,
        OFF#    OFF#,
        EMP#    EMP#,
        GRADE   GRADE }
      KEY { COURSE#, OFF#, EMP# }
      FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
                                    ON DELETE CASCADE
                                    ON UPDATE CASCADE
      FOREIGN KEY { EMP# } REFERENCES EMPLOYEE
                           ON DELETE CASCADE
                           ON UPDATE CASCADE ;
```

Points arising:

1. The (singleton) attribute sets {COURSE#} in TEACHER and {COURSE#} in ENROLLMENT could also be regarded as foreign keys, both of them referring to COURSE. However, if the referential constraints from TEACHER to OFFERING, ENROLLMENT to OFFERING, and OFFERING to COURSE are all properly maintained, the referential constraints from TEACHER to COURSE and ENROLLMENT to COURSE will be maintained automatically. See reference [9.11] for further discussion.

2. OFFERING is an example of a relvar that's simultaneously both referenced and referencing: There's a referential constraint *to* OFFERING from ENROLLMENT (also from TEACHER, as a matter of fact), and a referential constraint *from* OFFERING to COURSE:

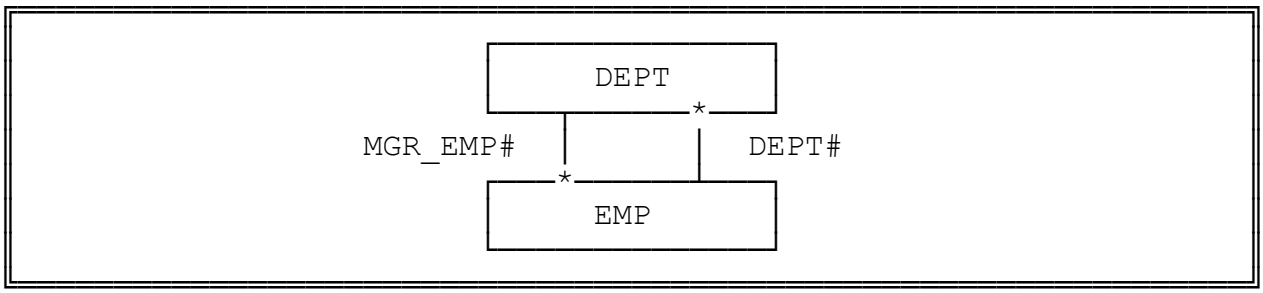
   ENROLLMENT ———* OFFERING ———* COURSE

3. Note that there are two distinct referential paths from ENROLLMENT to COURSE——one direct (foreign key {COURSE#} in ENROLLMENT), and the other indirect via OFFERING (foreign keys {COURSE#,OFF#} in ENROLLMENT and {COURSE#} in OFFERING):

However, the two paths aren't truly independent of one another (the upper path is implied by the combination of the lower two). For further discussion of this point, see reference [9.11] once again.

4. There are also two distinct referential paths from PREREQ to COURSE, but this time the two paths *are* truly independent (they have totally separate meanings). See reference [9.11] yet again.

9.8 The referential diagram is shown in the figure below. Note that the database involves a referential cycle (there's a referential path from each of the two relvars to itself). Apart from this consideration, the database definition is essentially straightforward. We omit the details.



9.9 We show just the relvar definitions (and those only in outline):

```
VAR EMP BASE RELATION
      { EMP# ... ,
        ...... ,
        JOB  ... }
      KEY { EMP# } ;

 VAR PGMR BASE RELATION
       { EMP# ... ,
         ...... ,
         LANG ... }
       KEY { EMP# }
       FOREIGN KEY { EMP# } REFERENCES EMP
               ON DELETE CASCADE
               ON UPDATE CASCADE ;
```

Points arising:

1. This example illustrates the point that a foreign key can also be a candidate key of its containing relvar. Relvar EMP contains all employees, and relvar PGMR contains just those employees that are programmers; thus, every employee number appearing in PGMR must also appear in EMP (but the converse isn't true). The primary key of PGMR is also a foreign key, referring to the primary key of EMP.

2. Note that there's another constraint that needs to be maintained in this example——namely, the constraint that a given employee will appear in PGMR if and only if the value of JOB for that employee is "Programmer." This constraint isn't a *referential* constraint, of course.

9.10 A candidate key with no attributes (a nullary or "empty key") is certainly possible. In particular, a nullary relvar——i.e., one that has no attributes, and hence one whose only legal values are DEE and DUM——"obviously" and necessarily has such a key. But a relvar doesn't have to be nullary itself in order to have a nullary key. However, it's at least true that if relvar *R* has a nullary key *NK,* then:

- *NK* is the *only* candidate key for *R,* because any nonempty set of attributes of *R* would include *NK* as a proper subset and would thus violate the irreducibility requirement for candidate keys.* (*NK* is therefore in fact the *primary* key, if a primary key must be chosen.)

———————

* Recall that the empty set is a subset of every set.

———————

- *R* is constrained to contain at most one tuple, because every tuple has the same value (namely, the 0-tuple) for *NK*. In other words, to say that *R* has a nullary key is to constrain *R* to contain at most one tuple, and such a constraint could certainly be useful in some circumstances [6.5].

Note that **Tutorial D** certainly does permit the declaration of such a relvar——for example:

```
VAR R BASE RELATION { ... }
    KEY { } ;
```

It also permits the declaration of a relvar with no attributes at all——i.e., a relvar whose only possible values are DEE and DUM:

```
VAR R BASE RELATION { }
    PRIMARY KEY { } ;
```

As an aside, we note that SQL doesn't support either of these possibilities.

9.11 Let *m* be the largest integer greater than or equal to *n*/2. *R* will have the maximum possible number of keys if either (a) every distinct set of *m* attributes is a key or (b) *n* is odd and every distinct set of *m*-1 attributes is a key. Either way, it follows that the maximum number of keys in *R* is *n*! / (*m*! * (*n*-*m*)!). *Note:* Relvars ELEMENT and MARRIAGE in Section 9.10 are both examples of relvars with the maximum possible number of keys; so is any nullary relvar. (If *n* = 0, the formula becomes 0!/(0!*0!), and 0! is 1.)

9.12 In many cases it's possible to make precise statements regarding *superkeys* only, rather than candidate keys as such.

a. Every key of *A* is a superkey for every restriction of *A*.

b. If the projection includes a key *K* of *A*, then *K* is a superkey for the projection. Otherwise all that can be said in general is that the combination of all attributes of the projection is a superkey for the projection.

c. Every combination *K* of a key *KA* of *A* and a key *KB* of *B* is a key for the product *A* TIMES *B*.

d. The combination of all attributes is a superkey for the union *A* UNION *B*.

e. Every key of *A* or *B* is a superkey for the intersection *A* INTERSECT *B*.

f. Every key of *A* is a superkey for the difference *A* MINUS *B*.

g. Every combination *K* of a key *KA* of *A* and a key *KB* of *B* is a superkey for the join *A* JOIN *B*. *Note:* In the special case where the joining attributes in *A* include a key of *A*, every key of *B* is a superkey for the join.

h. Every key of *A* is a key for every extension of *A*.

i. Every key of *B* is a superkey for an arbitrary summarization of *A* "per *B*."

j. Every key of *A* is a superkey for the semijoin *A* SEMIJOIN *B*.

k. Every key of *A* is a superkey for the semidifference *A* SEMIMINUS *B*.

However, many of the foregoing statements can be refined somewhat in certain situations. For example:

- The combination {S#,P#,J#} isn't the only superkey for the restriction SPJ WHERE S# = S#('S1'), because the combination {P#,J#} is as well.

- If *A* has heading {*X,Y,Z*} and sole candidate key *X* and satisfies the functional dependency $Y \rightarrow Z$ (see Chapter 11), then *Y* is a superkey for the projection of *A* over *Y* and *Z*.

- If *A* and *B* are both restrictions of *C*, then every key of *C* is a superkey for *A* UNION *B*.

This whole question of ***key inference*** is discussed in some detail in reference [11.7].

9.13 Clearly, if a ***candidate*** key can be empty, then so can a matching ***foreign*** key——and nullary foreign keys, like nullary candidate keys, can certainly be useful on occasion. See reference [6.5] for a detailed discussion.

9.14 Note first that SQL doesn't support type constraints, as such, at all. Part a. of the exercise thus can't be solved directly. However, we can keep the legal city names in a base table and use foreign keys to ensure that no other table ever includes a city name that isn't one of the legal ones.* Analogous remarks apply to part b. We omit further details here.

———————

* We could also use SQL-style "domains" [4.20].

———————

c.
```
CREATE ASSERTION SQL_C CHECK
      ( P.COLOR <> COLOR ( 'Red') OR
        P.WEIGHT < WEIGHT ( 50.0 ) ) ;
```

Here and throughout the rest of these answers we choose to use "assertions" rather than "base table check constraints."

d. CREATE ASSERTION SQL_D CHECK

```
        ( NOT EXISTS ( SELECT * FROM J AS JX WHERE
               EXISTS ( SELECT * FROM J AS JY WHERE
                        ( JX.J# <> JY.J# AND
                       JX.CITY = JY.CITY ) ) ) ) ;
```

e.
```
CREATE ASSERTION SQL_E CHECK
        ( ( SELECT COUNT(*) FROM S
            WHERE  S.CITY = 'Athens' ) ≤ 1 ) ;
```

f.
```
CREATE ASSERTION SQL_F CHECK
        ( NOT EXISTS ( SELECT *
                         FROM   SPJ AS SPJX
                       WHERE  SPJX.QTY > 2 *
                            ( SELECT AVG ( SPJY.QTY )
                              FROM   SPJ AS SPJY ) ) ) ;
```

g.
```
CREATE ASSERTION SQL_G CHECK
        ( NOT EXISTS ( SELECT * FROM S SX WHERE
             EXISTS ( SELECT * FROM S SY WHERE
                       SX.STATUS = ( SELECT MAX ( S.STATUS )
                                     FROM   S ) AND
                       SY.STATUS = ( SELECT MIN ( S.STATUS )
                                     FROM   S ) AND
                       SX.STATUS <> SY.STATUS AND
                       SX.CITY = SY.CITY ) ) ) ;
```

h.
```
CREATE ASSERTION SQL_H CHECK
        ( NOT EXISTS ( SELECT * FROM J WHERE
          NOT EXISTS ( SELECT * FROM S WHERE
                       S.CITY = J.CITY ) ) ) ;
```

i.
```
CREATE ASSERTION SQL_I CHECK
        ( NOT EXISTS ( SELECT * FROM J WHERE
          NOT EXISTS ( SELECT * FROM S WHERE
                        S.CITY = J.CITY AND
                       EXISTS ( SELECT * FROM SPJ
                                WHERE  SPJ.S# = S.S#
                                AND    SPJ.J# = J.J# ) ) ) ) ;
```

j.
```
CREATE ASSERTION SQL_J CHECK
        ( NOT EXISTS ( SELECT * FROM P )
           OR EXISTS ( SELECT * FROM P
                       WHERE  P.COLOR = COLOR ( 'Red' ) ) ) ;
```

k.
```
CREATE ASSERTION SQL_K CHECK
      ( ( SELECT AVG ( S.STATUS ) FROM S ) > 19 ) ;
```

If the suppliers table is empty, the SQL AVG operator will (incorrectly!) return a null, the comparison will evaluate to

unknown, and the constraint will *not* be regarded as violated. See Chapter 19 for further explanation.

l.
```
CREATE ASSERTION SQL_L CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.CITY = 'London'
                      AND    NOT EXISTS
                         ( SELECT * FROM SPJ
                           WHERE  SPJ.S# = S.S#
                           AND    SPJ.P# = P# ( 'P2' ) ) ) ) ;
```

m.
```
CREATE ASSERTION SQL_M CHECK
     ( NOT EXISTS ( SELECT * FROM P
                    WHERE  P.COLOR = COLOR ( 'Red' ) )
        OR EXISTS ( SELECT * FROM P
                    WHERE  P.COLOR = COLOR ( 'Red' )
                    AND    P.WEIGHT < WEIGHT ( 50.0 ) ) ) ;
```

n.
```
CREATE ASSERTION SQL_N CHECK
    ( ( SELECT COUNT ( DISTINCT P# ) FROM SPJ
        WHERE  EXISTS ( SELECT * FROM S WHERE
                      ( S.S# = SPJ.S# AND
                        S.CITY = 'London' ) ) ) >
      ( SELECT COUNT ( DISTINCT P# ) FROM SPJ
        WHERE  EXISTS ( SELECT * FROM S WHERE
                      ( S.S# = SPJ.S# AND
                        S.CITY = 'Paris' ) ) ) ) ;
```

o.
```
CREATE ASSERTION SQL_O CHECK
    ( ( SELECT SUM ( SPJ.QTY ) FROM SPJ
        WHERE  ( SELECT S.CITY FROM S
                 WHERE  S.S# = SPJ.S# ) = 'London' ) >
      ( SELECT SUM ( SPJ.QTY ) FROM SPJ
        WHERE  ( SELECT S.CITY FROM S
                 WHERE  S.S# = SPJ.S# ) = 'Paris' ) ) ;
```

Note the use of two scalar subqueries in this example.

p. Can't be done directly (SQL doesn't support transition constraints). We could write a trigger, though. *No further answer provided.*

q. Same as p.

9.15 The answers are trivial syntactic variations on those already given for Exercises 9.7-9.9. *No further answer provided.*

9.16 No. An important exception is predicates of the form illustrated by this example: "*i* is an integer." This is a *membership* predicate; in fact, it's the type constraint for the

type INTEGER.* ***Note:*** If instead of treating INTEGER as a type we tried to define a relvar INTEGER to list all possible integers, what type would the single attribute I of that relvar be?

---

* At least, for an ***ideal*** type INTEGER; we ignore the fact that no real computer system is capable of directly representing all possible integers.

9.17 Suppose we were to define a relvar SC{S#,CITY} with predicate "Supplier S# does not have an office in city CITY." Suppose further that supplier S1 has an office in just ten cities; then the Closed World Assumption would imply that relvar SC must have ***N***-10 tuples for supplier S1, where ***N*** is the total number of cities in the world!

***** End of Chapter 9 *****

Chapter 10

V i e w s

Principal Sections

- What are views for?
- View retrievals
- View updates
- Snapshots (a digression)
- SQL facilities

General Remarks

From the user's perspective, there should be ***no discernible difference*** between views and base relvars (another place where SQL and the vendors fail to support the relational model adequately——though, to be fair, the principles of view updating in particular were not well understood or articulated until the mid 1990s). A view ***is*** a relvar. ***Note:*** I deliberately include views in Part II of the book to emphasize the fact that view functionality was always intended to be a key ingredient of the relational model (see reference [6.9], Chapter 2).

The emphasis on views is yet another feature that sets this book apart from its competitors. In fact, a quick survey of six leading database textbooks reveals that:

- Not one has a chapter devoted to views.

- What coverage there is tends to be SQL-specific and thus reflects the usual SQL flaws. One book even limits its coverage to a section within a chapter entitled "Advanced SQL" [***sic!***].

- None of the books displays much insight into what views are really all about. What's more, several of them quite explicitly equate base and stored relvars, thereby implicitly violating ***The Principle of Interchangeability*** (see Section 10.2). The following extract from one of the books was quoted in a footnote in Chapter 3, and is fairly typical: "[It] is important to make a distinction between stored relations, which are ***tables,*** and virtual relations, which are ***views ...*** When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term ***base relation*** or ***base table***."

Copyright (c) 2003 C. J. Date

From the perspective of the relational model, a view is a *window* into certain underlying data. Views are (at least conceptually) implemented by *substitution,* not *materialization,*[*] and are thus NEVER "out of synch" with the underlying data. Views work——more precisely, the substitution process works——precisely because of *closure*.

_____

[*] As indicated in the notes on Chapter 3, materialization might sometimes be used as an implementation mechanism, but at the level of the *model* the phrase "materialized view" is a contradiction in terms. See Section 10.5.

_____

10.2 What Are Views For?

The benefits of views are as follows (to summarize):

1. Shorthand ("canned queries"——see the discussion in the final subsection in this section of the chapter)

2. Database customizing or tailoring

3. Security (but security isn't the *raison d'être* for views, it's more of a bonus)

4. **Logical data independence** (the big one!)

Note that logical data independence implies that it must be possible to define integrity constraints——in particular, candidate and foreign key constraints——that refer to views instead of base relvars. (Does SQL support this notion?)

The Principle of Interchangeability (of base and derived relvars): *There must be no arbitrary and unnecessary distinctions between base and derived relvars*.

The Principle of Database Relativity: Which expressible database is considered to be the "real" one is arbitrary, so long as all such databases are information-equivalent.[*] From the user's point of view, in other words, all relvars are base relvars, by definition (except for views explicitly defined by the user in question, as a shorthand).

———————

* Here's a loose definition: Two databases *A* and *B* are **information-equivalent** if and only if every relation that can be derived from *A* can also be derived from *B* and *vice versa*.

———————

10.3 View Retrievals

Basically self-explanatory. Regarding the final bulleted paragraph in this section (on cases in commercial products where the substitution procedure doesn't work), see Exercise 10.16 at the end of the chapter.

10.4 View Updates

The material of this section isn't included in other books (it's critically dependent on the idea of relvar predicates, also not covered in other books).* The section's rather long, and it might be desirable just to skim some of the examples (on the other hand, at least one reviewer of the previous edition suggested adding extra ones!).

———————

* You might want to note also that it's the subject of a recent US patent application (not by me)——see reference [10.11].

———————

All views are updatable (barring integrity constraint violations, of course).

Explain the basic idea of **predicate inference**. Discuss *The Golden Rule* again——*no relvar must ever be allowed to violate its own predicate*——and the various principles identified in this section:

1. View updatability is a semantic issue.

2. The updating mechanism must work for base relvars too (so we're really talking about a theory of updating in general, not just updating views specifically).

3. Preserve symmetry.

4. Take triggered procedures (including referential actions) into account.

5. UPDATE is shorthand for a DELETE-then-INSERT sequence (loosely speaking).

6. INSERTs map to INSERTs and DELETEs map to DELETEs.

7. The mechanism must be recursive.

8. We can't assume the database is well designed.

9. Each kind of view should support both INSERT and DELETE.

10. INSERT and DELETE should be inverses.

Recall that updates (as well as retrievals) are always **set-level,** though for simplicity the examples in the book mostly assume a set of cardinality one (i.e., a set that contains just one tuple).

Union/intersect/difference: These subsections should be self-explanatory. *Note:* Examples to illustrate various specific rules are left in the book as an exercise. *No answer provided.*

Restrict: Also self-explanatory. *Note:* The subsection says: "An attempt to update the LS tuple (S1,Smith,20,London) to (S6,Green,20,London) will succeed. An attempt to update that same tuple (S1,Smith,20,London) to either (S2,Smith,20,London) or (S1,Smith,20,Athens) will fail (why, exactly, in each case?)." *Answer:* The first fails because it violates the constraint (part of the relvar predicate for LS) that {S#} is a key. The second fails because it violates the constraint (also part of the relvar predicate for LS) that the CITY value must be London.

Project: Again fairly self-explanatory (though project is a little trickier than the operators covered prior to this point). *Note:* The subsection says: "An attempt to update the SC tuple (S1,London) ... to (S2,London) will fail (why, exactly?)." *Answer:* Because it violates the {S#} key constraint.

The subsection also says: "Consideration of the case in which the projection does not include a key of the underlying relvar (e.g., the projection of relvar S over STATUS and CITY) is left as an exercise." *Answer:* INSERTs probably fail (because probably there's no default defined for attribute S# in relvar S). DELETEs delete all tuples from S that have the same STATUS and CITY values as (any of) the tuple(s) specified for deletion from the projection. UPDATEs update all tuples in S that have the same

STATUS and CITY values as (any of) the tuple(s) specified for update in the projection.

Extend: Self-explanatory again. ***Note:*** The subsection says: "An attempt to insert the tuple (P7,Cog,Red,12,Paris,5449) will fail (why?) ... An attempt to insert the tuple (P1,Cog,Red,12,Paris,5448) will fail (why?)." ***Answer:*** The first fails because it violates the constraint that the GMWT value must be 454 times the WEIGHT value. The second fails because it violates the {P#} key uniqueness constraint.

The subsection also says: "An attempt to update [the tuple for P1] to one for P2 (with all other values unchanged) or to one in which the GMWT value is not equal to 454 times the WEIGHT value will fail (in each case, why?)." ***Answer:*** The first fails because it violates the {P#} key uniqueness constraint. The second fails because it violates the constraint that the GMWT value must be 454 times the WEIGHT value.

Join: The argument in support of the position that join views (like all other views!) are always theoretically updatable is important. It hinges on the flaw in the usual argument that join views are generally ***not*** updatable. The flaw in question is as follows:

a. It's usually assumed that it's always possible to update an individual tuple of a base relvar independently of all other tuples in that base relvar.

b. However, that assumption is incorrect!

One point (not necessarily for airing in class, but instructors should at least be aware of it): There's an argument——at least a syntactic one, though possibly not one of real substance——that says that an attempt to perform a view update that doesn't have ***exactly*** the specified effect should be rejected. For example, a request to insert a single tuple into a certain join view might (if accepted) have the effect of adding several tuples to the view. If it does, then the system is effectively performing certain compensating actions under the covers (akin, somewhat, to cascade deletes). I'm a little suspicious of compensating actions in general ... In the example, a real system might require the INSERT operator to specify ***all*** of the tuples that are to be added to the view. But the book ignores this possibility, and so do these notes from this point forward.

The subsection also asks a series of questions, as follows:*

* The notes that follow answer these questions for INSERT and DELETE only, not UPDATE.

- What's the effect of the update rules on the join of the suppliers relvar S to itself over supplier numbers (only)? *Answer:* Observe first that the join in question will have to be expressed in terms of certain RENAMEs——e.g., as follows:

  S JOIN (S RENAME (SNAME AS X, STATUS AS Y, CITY AS Z))

  The result satisfies the constraint that, in any given tuple, SNAME = X AND STATUS = Y AND CITY = Z.

  a. Inserting the tuple (*s,sn,st,sc,sn,st,sc*) into the join is equivalent to inserting the tuple (*s,sn,st,sc*) into S.

  b. Deleting the tuple (*s,sn,st,sc,sn,st,sc*) from the join is equivalent to deleting the tuple (*s,sn,st,sc*) from S.

- Suppose we have another base relvar SR with attributes S# and REST, where S# identifies a supplier and REST identifies that supplier's favorite restaurant. Assume that not all suppliers in S appear in SR. What's the effect of the join update rules on S JOIN SR? *Answer:*

  a. Inserting the tuple (*s,sn,st,sc,rt*) into the join is equivalent to inserting the tuple (*s,sn,st,sc*) into S (unless it's already present) and inserting the tuple (*s,rt*) into SR (it mustn't already be present).

  b. Deleting the tuple (*s,sn,st,sc,rt*) from the join is equivalent to deleting the tuple (*s,sn,st,sc*) from S and deleting the tuple (*s,rt*) from SR.

- What difference would it make if some supplier could appear in SR and not in S? *Answer:* No effect on DELETE. The following slight revision applies to INSERT:

  a. Inserting the tuple (*s,sn,st,sc,rt*) into the join is equivalent to inserting the tuple (*s,sn,st,sc*) into S (unless it's already present) and inserting the tuple (*s,rt*) into SR (unless it's already present).

- An attempt to insert the tuple (S4,Clark,20,Athens,P6,100) into SSP will fail (why?). *Answer:* Key uniqueness violation on S (note that SSP is defined as S JOIN SP).

- An attempt to insert the tuple (S1,Smith,20,London,P1,400) into SSP will fail (why?). *Answer:* Key uniqueness violation on SP.

- An attempt to update the SSP tuple (S1,Smith,20,London,P1,300) to (S6,Smith,20,London,P1,300) will "succeed"——see the note below——and will have the effect of updating the S tuple (S1,Smith,20,London) to (S6,Smith,20,London) and the SP tuple (S1,P1,300) to (S6,P1,300).

  Note: Actually, the overall effect of this attempted update will depend on the foreign key UPDATE rule from shipments to suppliers. The details are left as an exercise. *Answer:* If the rule specifies RESTRICT the overall operation will fail. If it specifies CASCADE it will have the side effect of updating all other SP tuples (and hence SSP tuples) for supplier S1 as well.

- "Further examples are left as an exercise" (examples, that is, of updates to S JOIN P). *No answer provided.*

Other Operators: Mostly self-explanatory. *Note:* Given the view-defining expression——

```
SUMMARIZE SP BY { S# } ADD SUM ( QTY ) AS TOTQTY
```

——the subsection says: "An attempt to insert the tuple (S5,0) will fail (why, exactly?)." *Answer:* Because (probably) no default is defined for attribute P# in relvar SP; also because (possibly) attribute QTY in relvar SP does not accept zero values.

10.5 Snapshots (a digression)

As the book says, this section is something of a digression from the main theme of the chapter. It's included partly (a) because snapshots are becoming increasingly important in practice, thanks to the growing use of (asynchronous) replication and data warehouse products, and partly (b) to criticize the prevailing use of the term "materialized views" or——worse——just "views" to refer to things that aren't views at all. To quote:

(Begin quote)

At the time of writing, snapshots have come to be known——almost exclusively, in fact——not as snapshots at all but rather as **materialized views*** (see the "References and Bibliography" section in Chapter 22). However, this terminology is unfortunate in the extreme, and in this writer's opinion should be resisted, firmly.

Snapshots are *not* views. The whole point about views is that they're *not* materialized, at least so far as the model is concerned. (Whether they're in fact materialized under the covers is an implementation issue and has nothing to do with the model.) As far as the model is concerned, in other words, "materialized view" is a contradiction in terms——and yet (all too predictably) "materialized view" has become so ubiquitous that the unqualified term *view* has come to mean, almost always, a "materialized view" specifically! And so we no longer have a good term to use when we want to refer to a view in the original sense. Certainly we run a severe risk of being misunderstood when we use the unqualified term *view* for that purpose. In this book, however, we choose to take that risk; to be specific, we won't use the term "materialized view" at all (except when quoting from other sources), keeping the term *snapshot* for the concept in question, and we'll always use the unqualified term *view* in its original relational sense.

_____

* Some writers (not all) reserve the term *materialized view* to mean a snapshot that is guaranteed to be always up to date——i.e., one for which REFRESH ON EVERY UPDATE has been specified.

_____

(End quote)

The section also asks what the predicate is for the following snapshot:

```
VAR P2SC SNAPSHOT
     ( ( S JOIN SP ) WHERE P# = P# ( 'P2' ) ) { S#, CITY }
        REFRESH EVERY DAY ;
```

Answer: "Supplier S# supplies part P2, and is located in city CITY, **as of at most 24 hours ago**." In fact, almost all relvar predicates (not just snapshot predicates) ought really to include some kind of temporal qualifier like this one ("as of ..."), but that qualifier is usually implicit. See Chapter 23 for further discussion.

10.6 SQL Facilities

Mostly self-explanatory, except for the rules regarding view updating (see below). *Note:* A very careful explanation of both LOCAL and CASCADED forms of WITH CHECK OPTION——and in particular

the complicated business of how they interact with each other and with "WITHOUT CHECK OPTION"——can be found in reference [4.20]. It might or might not help to point out that CASCADED in this context "goes the opposite way" from CASCADE in (e.g.) a DELETE rule.

I'm not alone in thinking that SQL's rules regarding view updating are hard to understand; a recent book on SQL:1999 [4.28] spends ten full and very confusing (and possibly confused) pages on the subject and ***still*** essentially fails to answer the question of exactly which views are updatable. Just to remind you, here's the quote from the standard that I show (as a "horrible example") in the chapter:

(Begin quote)

[The] ***<query expression> QE1*** is ***updatable*** if and only if for every ***<query expression>*** or ***<query specification> QE2*** that is simply contained in ***QE1:***

a) ***QE1*** contains ***QE2*** without an intervening ***<non join query expression>*** that specifies UNION DISTINCT, EXCEPT ALL, or EXCEPT DISTINCT.

b) If ***QE1*** simply contains a ***<non join query expression> NJQE*** that specifies UNION ALL, then:

   i) ***NJQE*** immediately contains a ***<query expression> LO*** and a ***<query term> RO*** such that no leaf generally underlying table of ***LO*** is also a leaf generally underlying table of ***RO***.

   ii) For every column of ***NJQE,*** the underlying columns in the tables identified by ***LO*** and ***RO,*** respectively, are either both updatable or not updatable.

c) ***QE1*** contains ***QE2*** without an intervening ***<non join query term>*** that specifies INTERSECT.

d) ***QE2*** is updatable.

(End quote)

I hope that's all perfectly clear! To me it all looks like epicycles on epicycles. Instead of a single systematic approach to the problem, SQL essentially treats the entire issue as a mishmash of special cases. As the book says, (a) the rule quoted above is just one of many that have to be taken in combination in order to determine whether a given view is updatable; (b) the rules in question aren't all given in one place but are scattered over many different parts of the document; and (c) all of those rules rely on a variety of additional concepts and

constructs——e.g., updatable columns, leaf generally underlying tables, *<non join query term>*s——that are in turn defined in still further parts of the document. It's hopeless.

My pragmatic teaching suggestion is this: Just list the simple cases from near the end of the section (noting, however, that even these limited cases are treated incorrectly, thanks to SQL's lack of understanding of predicates, and in particular to the fact that it permits duplicate rows). You might also mention that SQL identifies four distinct cases: *updatable, potentially updatable, simply updatable,* or *insertable into* (?). Which aren't exactly very clear.

References and Bibliography

Reference [10.3] is the source of Codd's famous——or infamous——"twelve rules" for relational DBMSs. An analysis of those rules can be found in reference [6.7]; see also the annotation to reference [10.2]. Reference [10.6] is, regrettably, still the only widely available source for the view updating mechanism advocated in the present chapter.

Answers to Exercises

10.1
```
VAR LONDON_SUPPLIER VIEW
    ( S WHERE CITY = 'London' ) { ALL BUT CITY } ;
```

We omit the CITY attribute here because we know its value must be London for every supplier in the view. Observe, however, that this omission means that any INSERT on the view will necessarily fail (unless the default value for attribute CITY in the underlying suppliers relvar happens to be London). In other words, a view like this one probably can't support INSERT operations at all. Alternatively, we might consider the possibility of defining the default value for CITY *for tuples inserted via this view* to be London. This idea of **view-specific defaults** requires more study. (Of course, we can achieve this effect by means of triggers, as we saw in Chapter 9. However, a declarative solution is naturally to be preferred.)

10.2
```
VAR NON_COLOCATED VIEW
    ( S { S# } JOIN P { P# } ) MINUS ( S JOIN P ) { S#, P# } ;
```

We could replace the first JOIN here by TIMES if we liked.

10.3 The question here is: How should attribute QTY be defined in the SP view? The sensible answer seems to be that, for a given

S#-P# pair, it should be the ***sum*** of all SPJ.QTY values, taken over all J#'s for that S#-P# pair:

```
VAR SP VIEW
    SUMMARIZE SPJ BY { S#, P# } ADD SUM ( QTY ) AS QTY ;
```

10.4
```
VAR JC VIEW
  ( ( SPJ WHERE S# = S# ( 'S1' ) ) { J# } JOIN
    ( SPJ WHERE P# = P# ( 'P1' ) ) { J# } ) JOIN
                                          J { J#, CITY } ;
```

10.5 We don't bother to show the converted forms. However, we remark that c. will fail at run time, because the tuple presented for insertion doesn't satisfy the predicate for the view.

10.6 Again c. fails at run time, though for a different reason this time. First, the DBMS will include a default WEIGHT value, ***w*** say, in the tuple to be inserted, since the user hasn't provided a "real" WEIGHT value (in fact, of course, the user ***can't*** provide a "real" WEIGHT value). Second, it's extremely unlikely that whatever WT (not WEIGHT) value the user provides will be equal to ***w*** * 454——even if (as is ***not*** the case in the INSERT shown) that particular WT value happens to be greater than 6356.0. Thus, the tuple presented for insertion again fails to satisfy the predicate for the view. ***Note:*** It could be argued that the WEIGHT value in the tuple to be inserted should properly be set to the specified WT value ***divided*** by 454. This possibility requires more study.

10.7 We've numbered the following solutions as 10.7.***n,*** where ***n*** is the number of the original example in Section 10.1. We make our usual assumptions regarding range variables.

10.7.1
```
VAR REDPART VIEW
      { PX.P#, PX.PNAME, PX.WEIGHT AS WT, PX.CITY }
        WHERE PX.COLOR = COLOR ( 'Red' ) ;
```

10.7.2
```
VAR PQ VIEW
      { PX.P#,
        SUM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY } ;
```

10.7.3
```
VAR CITY_PAIR VIEW
      { SX.CITY AS SCITY, PX.CITY AS PCITY }
        WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                           SPX.P# = PX.P# ) ;
```

10.7.4
```
VAR HEAVY_REDPART VIEW
        RPX WHERE RPX.WT > WEIGHT ( 12.0 ) ;
```

RPX here is a range variable that ranges over REDPART.

Copyright (c) 2003 C. J. Date

10.8 Because the result of ORDER BY isn't a relation.

10.9 The following list of reasons is taken from reference [6.7]:

- If users are to interact with views instead of base relvars, then it's clear that those views should look to the user as much like base relvars as possible. Ideally, in fact, the user should not even have to know they *are* views, but should be able to treat them as if they actually were base relvars, thanks to *The Principle of Database Relativity*. And just as the user of a base relvar needs to know what candidate keys that base relvar has (in general), so the user of a view needs to know what candidate keys that view has (again, in general). Explicitly declaring those keys is the obvious way to make that information available.

- The DBMS might be unable to deduce candidate keys for itself (this is almost certainly the case with DBMSs on the market today). Explicit declarations are thus likely to be the only means available (to the DBA, that is) of informing the DBMS——as well as the user——of the existence of such keys.

- Even if the DBMS were able to deduce candidate keys for itself, explicit declarations would at least enable the system to check that its deductions and the DBA's explicit specifications were consistent.

- The DBA might have some knowledge that the DBMS doesn't, and might thus be able to improve on the DBMS's deductions. Reference [6.7] gives an example of this possibility.

And reference [12.3] offers another reason, which is essentially that such a facility would provide a simple and convenient way of stating certain important integrity constraints that could otherwise be stated only in a very circumlocutory fashion.

10.10 It's obviously impossible to provide a definitive answer to this question. We offer the following observations.

- Each view and each snapshot will have an entry in the catalog relvar RELVAR (see the answer to Exercise 6.16), with a RVKIND value of "View" or "Snapshot" as appropriate. (RVKIND here——"relvar kind"——is an attribute of the catalog relvar RELVAR.)

- Each view will also have an entry in a new catalog relvar, which we might as well call VIEW. That entry should include the relevant view-defining expression.

Copyright (c) 2003 C. J. Date

- Similarly, each snapshot will also have an entry in a new catalog relvar (SNAPSHOT). That entry should include the relevant defining expression. It should also include information regarding the snapshot refresh interval.

- Yet another catalog relvar will show which views and snapshots are defined in terms of which other relvars. Note that the structure of this relvar is somewhat similar to that of the PART_STRUCTURE relvar (see Fig. 4.6 in Chapter 4): Just as parts can contain other parts, so views and snapshots can be defined in terms of other views and snapshots. Note, therefore, that the points discussed in the answer to Exercise 8.7 are relevant here.

10.11 Yes!——but note the following. Suppose we replace the suppliers relvar S by two restrictions, SA and SB say, where SA is the suppliers in London and SB is the suppliers not in London. We can now define the union of SA and SB as a view called S. If we now try (through this view) to UPDATE a London supplier's city to something other than London, or a "nonLondon" supplier's city to London, the implementation must map that UPDATE to a DELETE on one of the two restrictions and an INSERT on the other. Now, the rules given in Section 10.4 do handle this case correctly——in fact, we (deliberately) *defined* UPDATE as a DELETE followed by an INSERT; however, there was a tacit assumption that the implementation would actually use an UPDATE, for efficiency reasons. This example shows that sometimes mapping an UPDATE to an UPDATE does not work; in fact, determining those cases in which it does work can be regarded as an optimization.

10.12 Yes!

10.13 Yes!

10.14 INSERT and DELETE will always be inverses of each other so long as (a) the database is designed in accordance with *The Principle of Orthogonal Design* (see Chapter 13, Section 13.6) *and* (b) the DBMS supports relvar predicates properly. If these conditions aren't satisfied, however, then it's possible they might not be inverses of each other after all. For example, if *A* and *B* are distinct base relvars, inserting tuple *t* into *V* = *A* INTERSECT *B* might cause *t* to be inserted into *A* only (because it's already present in *B*); subsequently deleting *t* from *V* will now cause *t* to be deleted from both *A* and *B*. (On the other hand, deleting *t* from *V* and then reinserting it will always preserve the *status quo*.) However, note carefully that such an asymmetry can arise only if *t* satisfies the predicate for *A* and yet isn't present in *A* in the first place.

Copyright (c) 2003 C. J. Date

10.15 We offer the following comments. First, the replacement process itself involves several steps, which might be summarized as follows. (This sequence of operations will be refined in a moment.)

```
/* define the new base relvars */

VAR SNC BASE RELATION
      { S# S#, SNAME NAME, CITY CHAR }
        KEY { S# } ;

VAR ST BASE RELATION
     { S# S#, STATUS INTEGER }
       KEY { S# } ;

/* copy the data to the new base relvars */

INSERT SNC S { S#, SNAME, CITY } ;

INSERT ST  S { S#, STATUS } ;

/* drop the old relvar */

DROP VAR S ;
```

Now we can create the desired view:

```
VAR S VIEW
      SNC JOIN ST ;
```

We now observe that each of the two S# attributes (in SNC and ST) constitutes a foreign key that references the other. Indeed, there's a strict one-to-one relationship between relvars SNC and ST, and so we run into a variety of "one-to-one" difficulties that have been discussed in some detail by this writer elsewhere [14.8].*

---

* Some of those difficulties might be alleviated if the system supported multiple assignment, a possibility not discussed in reference [14.8].

Note also that we must do something about the foreign key in relvar SP that references the old base relvar S. Clearly, it would be best if that foreign key could now be taken as referring to the view S instead;* if this is impossible (as indeed it

typically is in today's products), then it would be better to add a third projection of base relvar S to the database, as follows:

---

* Indeed, logical data independence is a strong argument in favor of allowing constraints in general to be defined for views as well as base relvars.

```
VAR SS BASE RELATION
     { S# S# } KEY { S# } ;

INSERT SS S { S# } ;
```

(In fact, this design is recommended in reference [9.11] for other reasons anyway.) We now change the definition of view S thus:

```
VAR S VIEW
      SS JOIN SNC JOIN ST ;
```

We also add the following foreign key specification to the definitions of relvars SNC and ST:

```
FOREIGN KEY { S# } REFERENCES SS
                  ON DELETE CASCADE
                  ON UPDATE CASCADE
```

Finally, we must change the specification for the foreign key {S#} in relvar SP to refer to SS instead of S.

10.16 Regarding part a. of this exercise, here's one example of a view retrieval that certainly does fail in some products at the time of writing. Consider the following SQL view definition:

```
CREATE VIEW PQ AS
       SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY
       FROM   SP
       GROUP  BY SP.P# ;
```

Consider also the following attempted query:

```
SELECT AVG ( PQ.TOTQTY ) AS PT
FROM   PQ ;
```

If we follow the simple substitution process explained in the body of the chapter (i.e., we try to replace references to the view

name by the expression that defines the view), we obtain something like the following:

```
SELECT AVG ( SUM ( SP.QTY ) ) AS PT
FROM   SP
GROUP  BY SP.P# ;
```

And this isn't a valid SELECT statement, because (as noted in the discussion following Example 8.6.7 in Chapter 8) SQL doesn't allow aggregate operators to be nested in this fashion.

Here's another example of a query against the same view PQ that also fails in some products for much the same reason:

```
SELECT PQ.P#
FROM   PQ
WHERE  PQ.TOTQTY > 500 ;
```

Precisely because of the problem illustrated by these examples, incidentally, some products——IBM's DB2 is a case in point——sometimes physically materialize the view (instead of applying the more usual substitution procedure) and then execute the query against that materialized version. This technique will always work, of course, but it's liable to incur a performance penalty. Moreover, in the case of DB2 in particular, it's still the case that some retrievals on some views don't work; i.e., DB2 doesn't ***always*** use materialization if substitution doesn't work, nor is it easy to say exactly which cases work and which don't.

10.17 First, here's a definition of Design b. in terms of Design a.:

```
VAR SSP VIEW
    S JOIN SP ;

VAR XSS VIEW
    S MINUS ( S JOIN SP ) { S#, SNAME, STATUS, CITY } ;
```

And here's a definition of Design a. in terms of Design b.:

```
VAR S VIEW
    XSS UNION SSP { S#, SNAME, STATUS, CITY } ;

VAR SP VIEW
    SSP { S#, P#, QTY } ;
```

The applicable database constraints for the two designs can be stated as follows:

```
CONSTRAINT DESIGN_A
    IS_EMPTY ( SP { S# } MINUS S { S# } ) ;
```

```
    CONSTRAINT DESIGN_B
        IS_EMPTY ( SSP { S# } INTERSECT XSS { S# } ) ;
```

We remark in passing that——given that {S#} is a key for relvar S——constraint DESIGN_A here exemplifies another way of formulating a referential constraint.

Design a. is clearly superior, for reasons discussed in detail in Chapter 12.

10.18 We've numbered the following solutions as 10.18.*n*, where *n* is the number of the original exercise.

10.18.1
```
CREATE VIEW LONDON_SUPPLIER
       AS SELECT S.S#, S.SNAME, S.STATUS
          FROM   S
          WHERE  S.CITY = 'London' ;
```

10.18.2
```
CREATE VIEW NON_COLOCATED
       AS SELECT S.S#, P.P#
          FROM   S, P
          WHERE  S.CITY <> P.CITY ;
```

10.18.3
```
CREATE VIEW SP
       AS SELECT SPJ.S#, SPJ.P#, SUM ( SPJ.QTY ) AS QTY
          FROM   SPJ
          GROUP  BY SPJ.S#, SPJ.P# ;
```

10.18.4
```
CREATE VIEW JC
       AS SELECT J.J#, J.CITY
          FROM   J
          WHERE  J.J# IN ( SELECT SPJ.J#
                           FROM   SPJ
                           WHERE  SPJ.S# = S# ( 'S1' ) )
          AND    J.J# IN ( SELECT SPJ.J#
                           FROM   SPJ
                           WHERE  SPJ.P# = P# ( 'P1' ) ) ;
```

10.19 The criticism mentioned in this exercise is heard quite often. Here's a possible counterargument.

1. Loosely speaking, DELETE deletes a set of zero or more tuples from a specified relvar. For simplicity, let's assume that the set of tuples is always of cardinality one, and so we can talk, even more loosely, in terms of "deleting a tuple" from the relvar in question.

2. Tuples in relvars correspond to propositions (assumed by convention to be true ones). The propositions in question are

instantiations of the predicate corresponding to the relvar in question.

3. Hence, the real-world interpretation of the operation "delete a tuple" is "remove a proposition" (presumably because we don't believe it's true any more).

4. So the question becomes: What should the semantics of deleting a tuple from a "join view" be? There are two basic approaches we can take to this question:

   - The first is the one advocated in the present chapter. To spell it out, we take "remove a proposition" to mean "remove ***all portions of*** that proposition"——and we apply this rule uniformly across the board.

   - The other (which is effectively the one suggested by people who criticize the first approach) is to do different things depending on whether the join is one-to-one, many-to-one, or many-to-many. In other words, the suggestion is that in certain circumstances we can "remove a proposition" by effectively removing just some portion of it. Certainly such partial removal would mean that the database no longer states that the overall proposition is true.

   - But then we have to face up to questions such as: Does (e.g.) many-to-one mean a relationship that's ***inherently, necessarily*** many-to-one, or does it mean one that just happens to be many-to-one because of the actual values involved right now? Presumably the former——but we must be clear. And then does "inherently many-to-one" mean, specifically, a foreign-key-to-matching-candidate-key join, or are there other cases that are inherently many-to-one? (If there are, it would be wrong to make foreign key specifications the sole deciding factor.)

   - Whatever we do, we **mustn't** come up with rules that say the semantics of DELETE on a view depend on the ***syntax*** of how that view is defined and not on its ***semantics***. In particular, the rules for join must be consistent with those for intersect, difference, extend, and all of the other operators.

   - If we decide to define the semantics of DELETE on a join view in some special way, along the lines suggested above, then of course we'll have to define the semantics of INSERT in a special way too. (It's a fundamental principle that DELETE and INSERT should be inverses of each other, in the sense that "DELETE ***t*** FROM ***R***" followed immediately by "INSERT ***t*** INTO ***R***" should effectively be a no-op, as should "INSERT ***t*** INTO ***R***" followed immediately by "DELETE ***t*** FROM

R"——assuming the INSERTs and DELETEs all succeed, of course.)

- Again, if we decide to treat join views in some special way, then consistency dictates that we treat EACH AND EVERY relational operator in its own special way——special rules for union, special rules for divide, and so on. Everything becomes a special case (in fact, consistency dictates inconsistency!). This surely can't be a good idea. Of course, it's essentially what today's DBMSs all do, insofar as they address the problem at all.

The net of all this is that one simple rule that applies in all cases is surely the right way to go. Especially since, in the example of S JOIN SP, we can achieve the desired DELETE behavior by applying the DELETE direct to relvar SP instead of to the join view!

Of course, nothing in the foregoing argument precludes the possibility of placing logic in application code (sitting on top of the DBMS) that (a) allows the join to be displayed as a single table on the screen, (b) allows the end user to remove a row from that table somehow, and (c) implements that removal by doing a DELETE on relvar SP (only) under the covers. But we must avoid any suggestion that what the end user would be doing in such a scenario is a relational DELETE. It's a different operation (and the user would need to understand that fact, in general), it has different semantics, and it should be given a different name.

10.20 The relational model consists of five components:

1. An open-ended collection of **scalar types** (including in particular the type *boolean* or *truth value*)

   Comment: The scalar types can be system- or user-defined, in general; thus, a means must be available for users to define their own types (this requirement is implied, partly, by that "open-ended"). A means must therefore also be available for users to define their own operators, since types without operators are useless. The only built-in (i.e., system-defined) type we insist on is type BOOLEAN, but a real system will surely support integers, strings, etc., as well.

2. A **relation type generator** and an intended interpretation for relations of types generated thereby

   Comment: The relation type generator allows users to define their own relation types (in **Tutorial D**, the definition of a given relation type is, typically, bundled in with the definition of a relation variable *of* that type——there's no

separate "define relation type" operator, for reasons explained in detail in reference [3.3]). The intended interpretation for a given relation type is the predicate stuff.

3. Facilities for defining **relation variables** of such generated relation types

   Comment: Of course! Note that relation variables are the *only* variables allowed inside a relational database (*The Information Principle,* in effect).

4. A **relational assignment** operation for assigning relation values to such relation variables

   Comment: Variables are updatable by definition (that's what "variable" means); hence, every kind of variable is subject to assignment (that's how updating is done), and relation variables are no exception. Of course, INSERT, UPDATE, and DELETE shorthands are legal and indeed useful, but strictly speaking they *are* only shorthands.

5. An open-ended collection of generic **relational operators** for deriving relation values from other relation values

   Comment: These operators make up the relational algebra, and they're therefore built-in (though there's no inherent reason why users shouldn't be able to define additional ones). Note that the operators are *generic*——i.e., they apply to all possible relations, loosely speaking.

***** End of Chapter 10 *****

P A R T I I I

D A T A B A S E D E S I G N

The database design problem can be stated as follows: Given some body of data to be represented in a database, how do we decide on a suitable logical structure for that data? In other words, how do we decide what relvars should exist and what attributes they should have? (Of course, "design" here means *logical* or *conceptual* design specifically. The "right" way to do database design is to do a clean logical design first, and then, as a separate and subsequent step, to map that logical design into whatever physical structures the target DBMS happens to support. Logical design is a fit subject for a book of this nature, but physical design——though important——isn't.)

One significant point of difference between the treatment of design issues in this book and that found in some other books is the heavy emphasis on **data integrity** (the predicate stuff once again).

Database design is, sadly, still more of an art than a science. It's true that there are some scientific principles that can be brought to bear on the problem, and those principles are the subject of Chapters 11-13; unfortunately, however, there are numerous design issues that those principles just don't address at all. As a consequence, various design methodologies——some of them fairly rigorous, others less so, but all of them *ad hoc* to a degree——have been proposed, and such methodologies are the general subject of Chapter 14. (In fact, the principal focus of that chapter is on "E/R modeling," since that particular methodology is the one most widely used in practice——despite the fact that, at least in my opinion, it suffers from a variety of serious shortcomings. Some of those shortcomings are identified in the chapter.)

Note: See the preface for a discussion of my reasons for deferring the design chapters to what some might think is a fairly late part of the book.* Basically, I believe students aren't ready to design databases properly, or to appreciate design issues fully, until they have some understanding of what databases are all about and how they're meant to be used.

———————

Copyright (c) 2003 C. J. Date

* On the other hand, one reviewer of the previous edition suggested that Part III should be omitted entirely and made into a whole new book!

None of the chapters in this part of the book has a "SQL Facilities" section, for fairly obvious reasons.

***** End of Introduction to Part III *****

Chapter 11

Functional Dependencies

Principal Sections

- Basic definitions
- Trivial and nontrivial FDs
- Closure of a set of FDs
- Closure of a set of attributes
- Irreducible sets of FDs

General Remarks

This is the most formal chapter in the book. But it isn't *very* formal, and it isn't very long, and it can probably just be skimmed if the instructor doesn't want to get too deeply into formal proofs and the like. Indeed, the chapter is included, in part, just to show that there really is some mathematical rigor underlying relational database theory. But the focus of the book in general is, as noted in the preface, on **insight** and **understanding**, not on formalisms and algorithms (the latter can always be found in the references). Observe in particular that the book deliberately doesn't cover the theory of MVDs and JDs anywhere near as thoroughly as it does that of FDs.

Be that as it may, the proofs (etc.) in this chapter aren't really difficult, though we all know that formalism and precise terminology can be a little daunting to the average reader. However, the following ideas, at least, do need to be explained:

- What an **FD** is, and the fact that the interesting ones are those that hold "for all time," meaning they're **integrity constraints** (in fact, of course, the term "FD" is usually taken to refer to this latter case specifically).

- The left and right sides of an FD are **sets** of attributes.

- If K is a candidate key for R, then $K \rightarrow A$ holds for all attributes A of R.

- If R satisfies $X \rightarrow A$ and X is *not* a candidate key, then R will probably involve some **redundancy** (a hint that the FD notion might have a role to play in logical database

Copyright (c) 2003 C. J. Date

design——we'll be wanting to get rid of redundancy and therefore we'll be wanting to find ways to get rid of certain FDs).

- Some FDs imply others.

- Given a set of FDs, the complete set of FDs implied by the given set can be found by means of **Armstrong's inference rules** or **axioms** (the rules should at least be mentioned, and perhaps briefly illustrated, but they don't need to be exhaustively discussed).

11.2 Basic Definitions / 11.3 Trivial and Nontrivial FDs / 11.4 Closure of a Set of FDs / 11.5 Closure of a Set of Attributes / 11.6 Irreducible Sets of FDs

The material of these sections can be summarized as follows:

- First of all, every relvar necessarily satisfies certain **trivial** FDs (an FD is trivial if and only if the right side is a subset——not necessarily a proper subset, of course——of the left side).

- Given a set *S* of FDs, the **closure** S^+ of that set is the set of all FDs implied by the FDs in *S*. Armstrong's inference rules provide a **sound** and **complete** basis for computing S^+ from *S* (though we usually don't actually perform that computation). Several other useful rules can easily be derived from Armstrong's rules (see the exercises).

- Given a set *Z* of attributes of relvar *R* and a set *S* of FDs that hold for *R*, the **closure** Z^+ of *Z* under *S* is the set of all attributes *A* of *R* such that the FD $Z \rightarrow A$ is a member of S^+ (i.e., such that the FD $Z \rightarrow A$ is implied by the FDs in *S*). If and only if Z^+ is all of the attributes of *R*, *Z* is a **superkey** for *R* (and a **candidate key** is an irreducible superkey). There's a simple algorithm for computing Z^+ from *Z* and *S*, and hence a simple way of determining whether a given FD $X \rightarrow Y$ is a member of S^+ ($X \rightarrow Y$ is a member of S^+ if and only if *Y* is a subset of X^+).

- Two sets of FDs *S1* and *S2* are **equivalent** if and only if they're **covers** for each other, i.e., if and only if $S1^+ = S2^+$. Every set of FDs is equivalent to at least one **irreducible** set. A set of FDs is irreducible if and only if all three of the following are true:

a. Every FD in the set has a singleton right side.

b. No FD in the set can be discarded without changing the closure of the set.

c. No attribute can be discarded from the left side of any FD in the set without changing the closure of the set.

If *I* is an irreducible set equivalent to *S*, enforcing the FDs in *I* will automatically enforce the FDs in *S*.

The sections also contain three inline exercises:

- Check that the FDs stated to hold in the relation in Fig. 11.1 do in fact hold. *Answer:* Here, of course, we're talking about FDs that happen to hold in a specific relation *value*, not ones that hold for all time. The exercise is trivial. *No further answer provided.*

- State the complete set of FDs satisfied by relvar SCP. *Answer:* The most important ones are clearly:

  { S#, P# } → QTY
  S# → CITY

  There are 83 additional FDs (!) implied by these two (i.e., the closure consists of 85 FDs in total).

- Prove the algorithm given in Fig. 11.2 is correct. *No answer provided.*

Answers to Exercises

11.1 (a) An FD is basically a statement of the form $A \rightarrow B$, where *A* and *B* are each subsets of the set of attributes of *R*. Given that a set of *n* elements has 2^n possible subsets, it follows that each of *A* and *B* has 2^n possible values, and hence an upper limit on the number of possible FDs in *R* is 2^{2n}. (b) Every tuple *t* of *R* has the same value (namely, the 0-tuple) for that subtuple of *t* that corresponds to the empty set of attributes. If *B* is empty, therefore, the FD $A \rightarrow B$ is trivially true for all possible sets *A* of attributes of *R;* in fact, it's a *trivial* FD, in the sense of that term as defined in Section 11.3, and it isn't very interesting.* On the other hand, if *A* is empty, the FD $A \rightarrow B$ means all tuples of *R* have the same value for *B* (since they certainly all have the same value for *A*). And if *B* in turn is "all of the attributes of *R*"——i.e., if *R* has an empty key——then *R*

is constrained to contain at most one tuple (for further discussion, see the answer to Exercise 9.10).

---

* If *A* is empty as well, the FD degenerates to {} → {}, which has some claim to being "the least momentous observation that can be made in Relationland" [6.5].

11.2 The rules are ***sound*** in the sense that, given a set *S* of FDs, FDs not implied by *S* can't be derived from *S* using the rules. They're ***complete*** in the sense that all FDs implied by *S* can be so derived.

11.3 The ***reflexivity*** rule states that if *B* is a subset of *A*, then *A* → *B*. ***Proof:*** Let the relvar in question be *R*, and let *t1* and *t2* be any two tuples of *R* that agree on *A*. Then certainly *t1* and *t2* agree on *B*. Hence *A* → *B*.

The ***augmentation*** rule states that if *A* → *B*, then *AC* → *BC*. ***Proof:*** Again let the relvar in question be *R*, and let *t1* and *t2* be any two tuples of *R* that agree on *AC*. Then certainly *t1* and *t2* agree on *C*. They also agree on *A*, and therefore on *B*, because *A* → *B*. Hence they agree on *BC*. Hence *AC* → *BC*.

The ***transitivity*** rule states that if *A* → *B* and *B* → *C*, then *A* → *C*. ***Proof:*** Once again let the relvar in question be *R*, and let *t1* and *t2* be any two tuples of *R* that agree on *A*. Then *t1* and *t2* agree on *B*, because *A* → *B*. Hence they also agree on *C*, because *B* → *C*. Hence *A* → *C*.

11.4 The ***self-determination*** rule states that *A* → *A*. ***Proof:*** Immediate, by reflexivity.

The ***decomposition*** rule states that if *A* → *BC*, then *A* → *B* and *A* → *C*. ***Proof:*** *A* → *BC* (given) and *BC* → *B* by reflexivity. Hence *A* → *B* by transitivity (and likewise for *A* → *C*).

The ***union*** rule states that if *A* → *B* and *A* → *C*, then *A* → *BC*. ***Proof:*** *A* → *B* (given), hence *A* → *BA* by augmentation; also, *A* → *C* (given), hence *BA* → *BC* by augmentation. Hence *A* → *BC* by transitivity.

The *composition* rule states that if $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$. *Proof:* $A \rightarrow B$ (given), hence $AC \rightarrow BC$ by augmentation; likewise, $C \rightarrow D$ (given), hence $BC \rightarrow BD$ by augmentation. Hence $AC \rightarrow BD$ by transitivity.

11.5 This proof requires intersection and difference, as well as union, of sets of attributes; we therefore show all three operators explicitly, union included, in the proof. (By contrast, previous proofs used simple concatenation of attributes to represent union.)

1.	$A \rightarrow B$	(given)
2.	$C \rightarrow D$	(given)
3.	$A \rightarrow B \cap C$	(joint dependence, 1)
4.	$C - B \rightarrow C - B$	(self-determination)
5.	$A \cup (C - B) \rightarrow (B \cap C) \cup (C - B)$	(composition, 3, 4)
6.	$A \cup (C - B) \rightarrow C$	(simplifying 5)
7.	$A \cup (C - B) \rightarrow D$	(transitivity, 6, 2)
8.	$A \cup (C - B) \rightarrow B \cup D$	(composition, 1, 7)

This completes the proof.

The rules used in the proof are as indicated in the comments. The following rules are all special cases of Darwen's theorem: union, transitivity, composition, and augmentation. So too is the following useful rule:

- If $A \rightarrow B$ and $AB \rightarrow C$, then $A \rightarrow C$.

11.6 (a) The closure of a set of FDs is the set of all FDs that are implied by the given set. (b) The closure of a set of attributes is the set of all attributes that are functionally dependent on the given set.

11.7 The complete set of FDs——i.e., the closure——for relvar SP is as follows:

```
{ S#, P#, QTY } → { S#, P#, QTY }
{ S#, P#, QTY } → { S#, P# }
{ S#, P#, QTY } → { P#, QTY }
{ S#, P#, QTY } → { S#, QTY }
{ S#, P#, QTY } → { S# }
{ S#, P#, QTY } → { P# }
{ S#, P#, QTY } → { QTY }
{ S#, P#, QTY } → { }
```

```
{ S#, P# }      → { S#, P#, QTY }
{ S#, P# }      → { S#, P# }
{ S#, P# }      → { P#, QTY }
{ S#, P# }      → { S#, QTY }
{ S#, P# }      → { S# }
{ S#, P# }      → { P# }
{ S#, P# }      → { QTY }
{ S#, P# }      → { }

{ P#, QTY }     → { P#, QTY }
{ P#, QTY }     → { P# }
{ P#, QTY }     → { QTY }
{ P#, QTY }     → { }

{ S#, QTY }     → { S#, QTY }
{ S#, QTY }     → { S# }
{ S#, QTY }     → { QTY }
{ S#, QTY }     → { }

{ S# }          → { S# }
{ S# }          → { }

{ P# }          → { P# }
{ P# }          → { }

{ QTY }         → { QTY }
{ QTY }         → { }

{ }             → { }
```

11.8 $\{A,C\}^+ = \{A,B,C,D,E\}$. The answer to the second part of the question is yes.

11.9 Two sets *S1* and *S2* of FDs are equivalent if and only if they have the same closure.

11.10 A set of FDs is irreducible if and only if all three of the following properties hold:

- Every FD has a singleton right side.
- No FD can be discarded without changing the closure.
- No attribute can be discarded from the left side of any FD without changing the closure.

Copyright (c) 2003 C. J. Date

11.11 They're equivalent. Let's number the FDs of the first set as follows:

1. $A \rightarrow B$
2. $AB \rightarrow C$
3. $D \rightarrow AC$
4. $D \rightarrow E$

Now, 3 can be replaced by:

3. $D \rightarrow A$ and $D \rightarrow C$

Next, 1 and 2 together imply that 2 can be replaced by:

2. $A \rightarrow C$

But now we have $D \rightarrow A$ and $A \rightarrow C$, so $D \rightarrow C$ is implied (by transitivity) and so can be dropped, leaving:

3. $D \rightarrow A$

The first set of FDs is thus equivalent to the following irreducible set:

$A \rightarrow B$
$A \rightarrow C$
$D \rightarrow A$
$D \rightarrow E$

The second given set of FDs

$A \rightarrow BC$
$D \rightarrow AE$

is clearly also equivalent to this irreducible set. Thus, the two given sets are equivalent.

11.12 The first step is to rewrite the given set such that every FD has a singleton right side:

1. $AB \rightarrow C$
2. $C \rightarrow A$
3. $BC \rightarrow D$
4. $ACD \rightarrow B$
5. $BE \rightarrow C$
6. $CE \rightarrow A$
7. $CE \rightarrow F$

Copyright (c) 2003 C. J. Date

8. $CF \rightarrow B$
9. $CF \rightarrow D$
11. $D \rightarrow E$
11. $D \rightarrow F$

Now:

- 2 implies 6, so we can drop 6.

- 8 implies $CF \rightarrow BC$ (by augmentation), which with 3 implies $CF \rightarrow D$ (by transitivity), so we can drop 11.

- 8 implies $ACF \rightarrow AB$ (by augmentation), and 11 implies $ACD \rightarrow ACF$ (by augmentation), and so $ACD \rightarrow AB$ (by transitivity), and so $ACD \rightarrow B$ (by decomposition), so we can drop 4.

No further reductions are possible, and so we're left with the following irreducible set:

$AB \rightarrow C$
$C \rightarrow A$
$BC \rightarrow D$
$BE \rightarrow C$
$CE \rightarrow F$
$CF \rightarrow B$
$D \rightarrow E$
$D \rightarrow F$

Alternatively:

- 2 implies $CD \rightarrow ACD$ (by composition), which with 4 implies $CD \rightarrow B$ (by transitivity), so we can replace 4 by $CD \rightarrow B$.

- 2 implies 6, so we can drop 6 (as before).

- 2 and 10 imply $CF \rightarrow AD$ (by composition), which implies $CF \rightarrow ADC$ (by augmentation), which with (the original) 4 implies $CF \rightarrow B$ (by transitivity), so we can drop 8.

No further reductions are possible, and so we're left with the following irreducible set:

$AB \rightarrow C$
$C \rightarrow A$
$BC \rightarrow D$
$CD \rightarrow B$

BE → *C*
CE → *F*
CF → *D*
D → *E*
D → *F*

Observe, therefore, that there are two distinct irreducible equivalents for the original set of FDs.

11.13 FDs: *No answer provided.* Candidate keys: *L*, *DPC*, and *DPT*.

11.14 Abbreviating NAME, STREET, CITY, STATE, and ZIP* to *N*, *R*, *C*, *T*, and *Z*, respectively, we have:

N → *RCT* *RCT* → *Z* *Z* → *CT*

An obviously equivalent irreducible set is:

N → *R* *N* → *C* *N* → *T* *RCT* → *Z* *Z* → *C* *Z* → *T*

The only candidate key is *N*.

---

* By the way, did you know that ZIP is an acronym? It stands for *zoning improvement program*.

11.15 No! In particular, the FD *Z* → *CT* doesn't hold (though it "almost does"). If it did hold, it would mean that distinct city and state combinations always have distinct zip codes—but there are exceptions; for example, the cities of Jenner and Fort Ross in California both have zip code 95450.

11.16 We don't give a full answer to this exercise, but content ourselves with the following observations. First, the set is clearly not irreducible, since *C* → *J* and *CJ* → *I* together imply *C* → *I*. Second, an obvious *superkey* is {*A*,*B*,*C*,*D*,*G*,*J*} (i.e., the set of all attributes mentioned on the left sides of the given FDs). We can eliminate *J* from this set because *C* → *J*, and we can eliminate *G* because *AB* → *G*. Since none of *A*, *B*, *C*, *D* appears on the right side of any of the given FDs, it follows that {*A*,*B*,*C*,*D*} is a candidate key.

***** End of Chapter 11 *****

Chapter 12

Further Normalization I: 1NF, 2NF, 3NF, BCNF

Principal Sections

- Nonloss decomposition and FDs
- 1NF, 2NF, 3NF
- FD preservation
- BCNF
- A note on RVAs

General Remarks

This chapter is concerned with FDs as an aid to database design; don't skip it. The treatment is deliberately not as formal as that of the preceding chapter. Note in particular the following caveat from the beginning of Section 12.3:

(Begin quote)

Throughout this section [on 1NF, 2NF, and 3NF], we assume for simplicity that each relvar has exactly one candidate key, which we further assume is the primary key. These assumptions are reflected in our definitions, which ... aren't very rigorous. The case of a relvar having more than one candidate key is discussed in Section 12.5.

(End quote)

A little bit of history: The first three normal forms were originally defined by Ted Codd, and they weren't too hard to understand. But then more and more researchers (Ted Codd, Raymond Boyce, Ron Fagin, others) began to define more and more new normal forms——Boyce/Codd, 4th, 5th, as well as some others not shown in Fig. 12.2——and people began to panic: Where's this all going to end? Will there be a 6th, a 7th, an 8th, a 9th, a 10th, ... normal form? Will there ever be an end to this progression? Well, I'm pleased to be able to tell you that there ***is*** an end: Fifth normal form really is the ***final*** normal form——in a very special sense, which we'll get to in the next chapter.

Copyright (c) 2003 C. J. Date

The basic problem with a relvar that's less than fully normalized* is **redundancy**. Redundancy in turn leads to **"update anomalies."** Note the little piece of insight in the footnote near the beginning of Section 12.1:

(Begin quote)

Throughout this chapter and the next, it's necessary to assume (realistically enough!) that relvar predicates aren't being fully enforced——for if they were, [some of the update anomalies to be discussed] couldn't possibly arise ... One way to think about the normalization discipline is as follows: It helps structure the database in such a way as to make more single-tuple updates logically acceptable than would otherwise be the case (i.e., if the design weren't fully normalized). This goal is achieved because the relvar predicates are simpler if the design is fully normalized than they would be otherwise.

(End quote)

———————

* To jump ahead to Chapter 13 for a moment, a precise statement of what it means for relvar ***R*** to be "less than fully normalized" is that ***R*** satisfies a certain JD that's not implied by the candidate keys of ***R***. Of course, that JD might be an MVD or even an FD.

———————

Normalized and **1NF** mean *exactly the same thing*——though "normalized" is often used to mean some higher level of normalization (typically 3NF). All relvars are in 1NF (see Chapter 6 and/or the article "What Does First Normal Form Really Mean?" (in two parts), due to appear soon on the website *www.dbdebunk.com*. ***Note:*** In particular, this article contains an extended treatment of RVAs——more extensive than the treatment in the present chapter. I wouldn't suggest including such extensive treatment in a live class, but as an instructor you might want to be aware of some of the issues.

Full normalization isn't ***required*** but is STRONGLY recommended. Backing off from full normalization usually implies unforeseen problems (but might be necessary in today's products, given their weak logical/physical separation).

In practice we rarely apply the normalization procedure directly; rather, we use the ideas of normalization to verify that a design achieved in some other manner doesn't unintentionally violate normalization principles. But the normalization procedure does provide a convenient framework in which to describe those principles——so we adopt the useful fiction (for the purposes of this chapter only) that we are indeed carrying out the design process by applying that procedure.

12.2 Nonloss Decomposition and FDs

Explain **nonloss decomposition** (reversibility) and **Heath's theorem**. Stress the role of the ***projection*** and ***join*** operators. Discuss **left-irreducible FDs** (*aka* "full" FDs). Explain **FD diagrams**.

With regard to nonloss decomposition, note the discussion of the additional requirement that none of the projections is redundant in the (re)join: "For simplicity, let's agree from this point forward that this additional requirement is in fact always in force, barring explicit statements to the contrary."

A nice intuitive characterization of the normalization procedure (at least up to BCNF): It's a procedure for ***eliminating arrows that aren't arrows out of candidate keys***. Note that this characterization can be extended straightforwardly to deal with normalization up to 4NF and 5NF as well (see Chapter 13).

This section includes the following inline exercise:

> [If we replace S by two projections and then join those projections back together again,] we get back all of the tuples in the original S, [possibly] together with some additional "spurious" tuples; we can never get back anything ***less*** than the original S. ***Exercise:*** Prove this statement.

Answer: Let X and Y be the two projections, let the attributes common to X and Y be B, let the other attributes of X be A, and let the other attributes of Y be C (the [disjoint] union of A, B, and C is all of the attributes of S, of course). Let $t = (a,b,c)$ be a tuple in S. Then tuple $tx = (a,b)$ appears in X and tuple $ty = (b,c)$ appears in Y, whence tuple $t = (a,b,c)$ appears in the join of X and Y. ∎

The section also leaves as an exercise detailed consideration of how replacing SECOND by SC and CS overcomes certain update anomalies. ***Answer:***

- **INSERT:** We can insert the information that Rome has a status of 50, even though no supplier is currently located in Rome, by simply inserting the appropriate tuple into CS.

- **DELETE:** We can delete supplier S5 from SC without losing the information that Athens has status 30.

- **UPDATE:** In the revised structure, the status for a given city appears once, not many times, because there's precisely one tuple for a given city in CS (the primary key is {CITY}); in other words, the CITY-STATUS redundancy has been eliminated. Thus, we can change the status for London from 20 to 30 by changing it once and for all in the relevant CS tuple.

12.3 1NF, 2NF, 3NF

Mostly self-explanatory. Another nice intuitive characterization of the normalization procedure: It's an *unbundling* procedure——put logically separate information into separate relvars. Highlight the following "algorithms":

1. Given: *R* { *A, B, C, D* }
   PRIMARY KEY { *A, B* }
   /* assume *A* → *D* holds */

   Replace *R* by *R1* and *R2:*

   R1 { *A, D* }
   PRIMARY KEY { *A* }

   R2 { *A, B, C* }
   PRIMARY KEY { *A, B* }
   FOREIGN KEY { *A* } REFERENCES *R1*

2. Given: *R* { *A, B, C* }
   PRIMARY KEY { *A* }
   /* assume *B* → *C* holds */

   Replace *R* by *R1* and *R2:*

   R1 { *B, C* }
   PRIMARY KEY { *B* }

   R2 { *A, B* }
   PRIMARY KEY { *A* }
   FOREIGN KEY { *B* } REFERENCES *R1*

If you want to get into more formalism, see the algorithm at the end of Section 12.4 for obtaining 3NF (in an FD-preserving way).

Note that a given relvar can be said to be at a given level of normalization ***only with respect to a specified set of dependencies*** (but it's usual to ignore this point in informal contexts). E.g., the relvar

NADDR { NAME, STREET, CITY, STATE, ZIP }

can be regarded as fully normalized if the FD ZIP → { CITY, STATE } is of no interest and hence isn't mentioned. (Of course, that FD doesn't really hold in practice anyway, as we saw in the answers to the exercises in Chapter 11.)

12.4 FD Preservation

Like further normalization in general, FD preservation can be seen as a way of designing the database in such a manner as to simplify the integrity constraints that need to be stated and enforced.

The section includes the following: "Replacing SECOND by its two projections on {S#,STATUS} and {CITY,STATUS} isn't a valid decomposition, because it isn't nonloss. ***Exercise:*** Prove this statement." ***Answer:*** Given the usual sample data values, the join of these two projections clearly includes a tuple relating supplier S3 to the city Athens, yet no such tuple appears in the original S.

12.5 BCNF

BNCF is **the** normal form if FDs are the only kind of dependency considered; in some respects, therefore, 2NF and 3NF are of historical interest merely (though they can be pragmatically useful concepts in the practical business of database design). Presumably for this very reason, some textbooks go straight to BCNF and ignore 2NF and 3NF.

Regarding the SSP example: Students might object that SSP is not even in 2NF, because (e.g.) SNAME is not irreducibly dependent on the "primary" key {S#,P#}. (If nobody does object, then raise the objection yourself!) Explain that it ***is*** in 2NF (and 3NF) according to Codd's original definitions [11.6]——the definitions in Section 12.3 were deliberately somewhat simplified, and ignored the glitch in Codd's original definition. (Zaniolo's nice definition of 3NF, repeated below, is equivalent to Codd's original definition.)

Stress the point that BCNF (like all the other formal ideas discussed in this chapter and the next) are basically just *formalized common sense*——but formalizing common sense is a neat trick! (and not easy to do).

BCNF and FD preservation can be conflicting objectives (see the SJT example).

Zaniolo's nice definitions:

- **3NF:** *R* is in 3NF if and only if, for every FD $X \rightarrow A$ in *R*, at least one of the following is true:

  1. *X* contains *A* (so the FD is trivial).

  2. *X* is a superkey.

  3. *A* is contained in a candidate key of *R*.

- **BCNF:** As above, except (a) drop possibility 3 and (b) replace "3NF" by "BCNF" (of course). Possibility 3 is why SSP *is* in 3NF, incidentally (see above); it corresponds to the glitch in Codd's original definition.

Note that Zaniolo's definitions make it immediately obvious that (a) all BCNF relvars are in 3NF and (b) the converse isn't true (there do exist 3NF relvars that aren't in BCNF).

If you want to get into more formalism, see the algorithm at the end of this section for obtaining BCNF (albeit not necessarily in an FD-preserving way, given that BCNF and FD preservation can be conflicting objectives, as we already know).

In its discussion of the SJT example (in which SJT is replaced by the two projections ST{S,T} and TJ{T,J}), this section includes the following: "Show the values of these two relvars corresponding to the data of Fig. 12.14; draw a corresponding FD diagram; prove that the two projections are indeed in BCNF (what are the candidate keys?); and check that the decomposition does in fact avoid the anomalies." *Answer:* ST satisfies no nontrivial FDs at all; TJ has {T} as its sole key and satisfies no nontrivial FDs except for the FD $\{T\} \rightarrow \{J\}$; both are therefore in BCNF. *No answer provided for the rest of the exercise.*

In its discussion of the EXAM example, the section includes the following: "However, EXAM is in BCNF, because the candidate keys are the only determinants, and update anomalies such as those discussed earlier in the chapter don't occur with this relvar. *Exercise:* Check this claim." *Answer:* It's easy to see that

"update anomalies such as those discussed earlier in the chapter" don't occur. But others can! For example, deleting the tuple {S:Smith,J:Math,P:5} will "leave a gap," in the sense that now nobody comes 5th in the class list with respect to Math (in other words, a certain integrity constraint has been violated). The EXAM example thus clearly illustrates the point that not all update anomalies can be eliminated by normalization (i.e., by taking projections). In fact, of course, normalization can eliminate ***precisely*** those anomalies that are caused by FDs or MVDs or JDs that aren't implied by keys——just those anomalies and no others.

12.6 A Note on RVAs

Possibly skip this section on a first pass. While RVAs are legal (see Chapter 6), they're usually contraindicated. (Of course, most textbooks——including earlier editions of this one——regard RVAs as illegal anyway. The section thus perhaps requires careful attention more by people who already know something about relational databases than it does by beginners.)

If you do cover this material, certainly point out the asymmetry (fundamental problem) and mention ***predicate complexity***. Here are the examples from the text. First, the (symmetric) queries——

1. Get S# for suppliers who supply part P1

2. Get P# for parts supplied by supplier S1

——have very different formulations:

```
1. ( SPQ WHERE TUPLE { P# P# ('P1') } ε PQ { P# } ) { S# }

2. ( ( SPQ WHERE S# = S# ('S1') ) UNGROUP PQ ) { P# }
```

Second, the (symmetric) updates——

1. Create a new shipment for supplier S6, part P5, quantity 500

2. Create a new shipment for supplier S2, part P5, quantity 500

——look like this:

```
1. INSERT SPQ RELATION
        { TUPLE { S# S# ('S6'),
                  PQ RELATION { TUPLE { P#  P# ('P5'),
                                        QTY QTY ( 500 ) } } } } ;
```

```
 2. UPDATE SPQ WHERE S# = S# ('S2')
        { INSERT PQ RELATION { TUPLE { P#  P# ('P5'),
                                       QTY QTY ( 500 ) } } } ;
```

Moreover, *all* of these formulations are significantly more complicated than their SP counterparts.

RVAs are thus usually contraindicated *in base relvars* (i.e., in logical DB designs). This doesn't mean they're contraindicated in derived relations or relvars, or *always* contraindicated even in base relvars.

By the way, relvar SPQ is in 5NF! (and thus certainly in BCNF).

Answers to Exercises

12.1 Heath's theorem states that if $R\{A,B,C\}$ satisfies the FD $A \rightarrow B$ (where A, B, and C are sets of attributes), then R is equal to the join of its projections $R1$ on $\{A,B\}$ and $R2$ on $\{A,C\}$. In the following proof of this theorem, we adopt our usual informal shorthand for tuples.

First we show that no tuple of R is lost by taking the projections and then joining those projections back together again. Let $(a,b,c) \in R$. Then $(a,b) \in R1$ and $(a,c) \in R2$, and so $(a,b,c) \in R1$ JOIN $R2$.

Next we show that every tuple of the join is indeed a tuple of R (i.e., the join doesn't generate any "spurious" tuples). Let $(a,b,c) \in R1$ JOIN $R2$. In order to generate such a tuple in the join, we must have $(a,b) \in R1$ and $(a,c) \in R2$. Hence there must exist a tuple $(a,b',c) \in R$ for some b', in order to generate the tuple $(a,c) \in R2$. We therefore must have $(a,b') \in R1$. Now we have $(a,b) \in R1$ and $(a,b') \in R1$; hence we must have $b = b'$, because $A \rightarrow B$. Hence $(a,b,c) \in R$.

The converse of Heath's theorem would state that if $R\{A,B,C\}$ is equal to the join of its projections on $\{A,B\}$ and on $\{A,C\}$, then R satisfies the FD $A \rightarrow B$. This statement is false. For example, Fig. 13.2 in the next chapter shows a relation that's certainly equal to the join of two of its projections and yet doesn't satisfy any (nontrivial) FDs at all.

12.2 The claim is almost but not quite valid. The following (pathological?) counterexample is taken from reference [6.5]. Consider the relvar

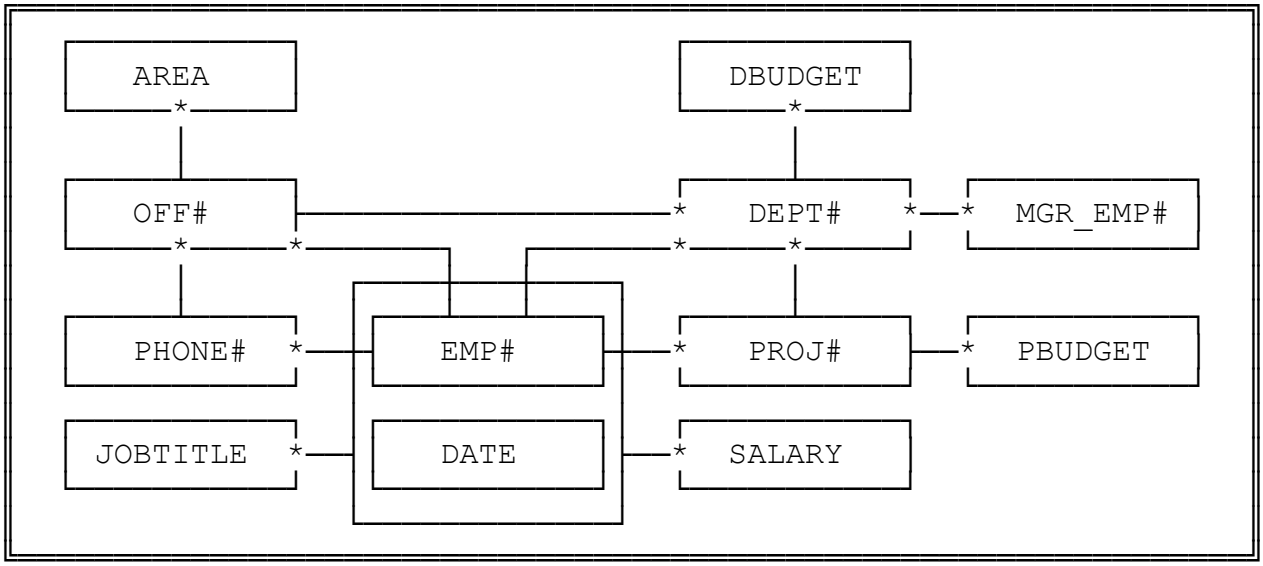
USA { COUNTRY, STATE }

(interpreted as "STATE is part of COUNTRY," where COUNTRY is the United States of America in every tuple). Then the FD

{ } → COUNTRY

holds in this relvar, and yet the empty set {} is not a candidate key. So USA isn't in BCNF (it can be nonloss-decomposed into its two unary projections——though whether it really should be further normalized in this way might be the subject of debate).

12.3 The figure below shows the most important FDs, both those implied by the wording of the exercise and those corresponding to reasonable semantic assumptions (stated explicitly below). The attribute names are intended to be self-explanatory.



Semantic assumptions:

- No employee is the manager of more than one department at a time.
- No employee works in more than one department at a time.
- No employee works on more than one project at a time.
- No employee has more than one office at a time.
- No employee has more than one phone at a time.
- No employee has more than one job at a time.

Copyright (c) 2003 C. J. Date

- No project is assigned to more than one department at a time.
- No office is assigned to more than one department at a time.
- Department numbers, employee numbers, project numbers, office numbers, and phone numbers are all "globally" unique.

Step 0: Establish initial relvar structure

Observe first that the original hierarchic structure can be regarded as a 1NF relvar DEPT0 with ***relation-valued attributes:***

```
DEPT0 { DEPT#, DBUDGET, MGR_EMP#, XEMP0, XPROJ0, XOFFICE0 }
      KEY { DEPT# }
      KEY { MGR_EMP# }
```

Attributes DEPT#, DBUDGET, and MGR_EMP# are self-explanatory, but attributes XEMP0, XPROJ0, and XOFFICE0 are relation-valued and do require a little more explanation:

- The XPROJ0 value within a given DEPT0 tuple is a relation with attributes PROJ# and PBUDGET.
- Likewise, the XOFFICE0 value within a given DEPT0 tuple is a relation with attributes OFF#, AREA, and (say) XPHONE0, where XPHONE0 is relation-valued in turn. XPHONE0 relations have just one attribute, PHONE#.
- Finally, the XEMP0 value within a given DEPT0 tuple is a relation with attributes EMP#, PROJ#, OFF#, PHONE#, and (say) XJOB0, where XJOB0 is relation-valued in turn. XJOB0 relations have attributes JOBTITLE and (say) XSALHIST0, where XSALHIST0 is once again relation-valued (XSALHIST0 relations have attributes DATE and SALARY).

The complete hierarchy can thus be represented by the following nested structure:

```
DEPT0 { DEPT#, DBUDGET, MGR_EMP#,
        XEMP0 { EMP#, PROJ#, OFF#, PHONE#,
                XJOB0 { JOBTITLE,
                        XSALHIST0 { DATE, SALARY } } },
        XPROJ0 { PROJ#, PBUDGET },
        XOFFICE0 { OFF#, AREA, XPHONE0 { PHONE# } } }
```

Note: Instead of attempting to show candidate keys, we've used ***italics*** here to indicate attributes that are at least "unique

within parent" (in fact, DEPT#, EMP#, PROJ#, OFF#, and PHONE# are, according to our stated assumptions, all *globally* unique).

Step 1: Eliminate relation-valued attributes

Now let's assume for simplicity that we wish every relvar to have a *primary* key specifically——i.e., we'll always designate one candidate key as primary for some reason (the reason isn't important here). In the case of DEPT0 in particular, let's choose {DEPT#} as the primary key (and so {MGR_EMP#} becomes an alternate key).

We now proceed to get rid of all of the relation-valued attributes in DEPT0, since as noted in Section 12.6 such attributes are usually undesirable:*

———————

* We remark that the procedure given here for eliminating RVAs amounts to repeatedly executing the UNGROUP operator (see Chapter 7, Section 7.9) until the desired result is obtained. Incidentally, the procedure as described also guarantees that any multi-valued dependencies (MVDs) that aren't FDs are eliminated too; as a consequence, the relvars we eventually wind up with are in fact in 4NF, not just BCNF (see Chapter 13).

———————

- For each RVA in DEPT0——i.e., attributes XEMP0, XPROJ0, and XOFFICE0——form a new relvar with attributes consisting of the attributes from the underlying relation type, together with the primary key of DEPT0. The primary key of each such relvar is the combination of the attribute that previously gave "uniqueness within parent," together with the primary key of DEPT0. (Note, however, that many of those "primary keys" will include attributes that are redundant for unique identification purposes and will be eliminated later in the overall reduction procedure.) Remove attributes XEMP0, XPROJ0, and XOFFICE0 from DEPT0.

- If any relvar *R* still includes any RVAs, perform an analogous sequence of operations on *R*.

We obtain the following collection of relvars, with (as indicated) all RVAs eliminated. Note, however, that while the resulting relvars are necessarily in 1NF (of course), they aren't necessarily in any higher normal form.

```
DEPT1 { DEPT#, DBUDGET, MGR_EMP# }
      PRIMARY KEY { DEPT# }
      ALTERNATE KEY { MGR_EMP# }

EMP1 { DEPT#, EMP#, PROJ#, OFF#, PHONE# }
     PRIMARY KEY { DEPT#, EMP# }

JOB1 { DEPT#, EMP#, JOBTITLE }
     PRIMARY KEY { DEPT#, EMP#, JOBTITLE }

SALHIST1 { DEPT#, EMP#, JOBTITLE, DATE, SALARY }
         PRIMARY KEY { DEPT#, EMP#, JOBTITLE, DATE }

PROJ1 { DEPT#, PROJ#, PBUDGET }
      PRIMARY KEY { DEPT#, PROJ# }

OFFICE1 { DEPT#, OFF#, AREA }
        PRIMARY KEY { DEPT#, OFF# }

PHONE1 { DEPT#, OFF#, PHONE# }
       PRIMARY KEY { DEPT#, OFF#, PHONE# }
```

Step 2: Reduce to 2NF

We now reduce the relvars produced in Step 1 to an equivalent collection of relvars in 2NF by eliminating any FDs that aren't irreducible. We consider the relvars one by one.

DEPT1: This relvar is already in 2NF.

EMP1: First observe that DEPT# is actually redundant as a component of the primary key for this relvar. We can take {EMP#} alone as the primary key, in which case the relvar is in 2NF as it stands.

JOB1: Again, DEPT# isn't needed as a component of the primary key. Since DEPT# is functionally dependent on EMP#, we have a nonkey attribute (DEPT#) that isn't irreducibly dependent on the primary key (the combination {EMP#,JOBTITLE}), and hence JOB1 isn't in 2NF. We can replace it by

```
JOB2A { EMP#, JOBTITLE }
      PRIMARY KEY { EMP#, JOBTITLE }
```

and

```
JOB2B { EMP#, DEPT# }
      PRIMARY KEY { EMP# }
```

However, JOB2A is a projection of SALHIST2 (see below), and JOB2B is a projection of EMP1 (renamed as EMP2 below), so both of these relvars can be discarded.

SALHIST1: As with JOB1, we can project away DEPT# entirely. Moreover, JOBTITLE isn't needed as a component of the primary key; we can take the combination {EMP#,DATE} as the primary key, to obtain the 2NF relvar

```
SALHIST2 { EMP#, DATE, JOBTITLE, SALARY }
               PRIMARY KEY { EMP#, DATE }
```

PROJ1: As with EMP1, we can consider DEPT# as a nonkey attribute; the relvar is then in 2NF as it stands.

OFFICE1: Similar remarks apply.

PHONE1: We can project away DEPT# entirely, since the relvar (DEPT#,OFF#) is a projection of OFFICE1 (renamed as OFFICE2 below). Also, OFF# is functionally dependent on PHONE#, so we can take {PHONE#} alone as the primary key, to obtain the 2NF relvar

```
PHONE2 { PHONE#, OFF# }
       PRIMARY KEY { PHONE# }
```

Note that this relvar isn't necessarily a projection of EMP2 (phones or offices might exist without being assigned to employees), so we can't discard it.

Hence our collection of 2NF relvars is

```
DEPT2 { DEPT#, DBUDGET, MGR_EMP# }
      PRIMARY KEY { DEPT# }
      ALTERNATE KEY { MGR_EMP# }

EMP2 { EMP#, DEPT#, PROJ#, OFF#, PHONE# }
     PRIMARY KEY { EMP# }

SALHIST2 { EMP#, DATE, JOBTITLE, SALARY }
         PRIMARY KEY { EMP#, DATE }

PROJ2 { PROJ#, DEPT#, PBUDGET }
      PRIMARY KEY { PROJ# }

OFFICE2 { OFF#, DEPT#, AREA }
        PRIMARY KEY { OFF# }

PHONE2 { PHONE#, OFF# }
       PRIMARY KEY { PHONE# }
```

Copyright (c) 2003 C. J. Date

Step 3: Reduce to 3NF

Now we reduce the 2NF relvars to an equivalent 3NF set by eliminating transitive FDs. The only 2NF relvar not already in 3NF is the relvar EMP2, in which OFF# and DEPT# are both transitively dependent on the primary key {EMP#}——OFF# via PHONE#, and DEPT# via PROJ# and also via OFF# (and hence via PHONE#). The 3NF relvars (projections) corresponding to EMP2 are

```
EMP3 { EMP#, PROJ#, PHONE# }
     PRIMARY KEY { EMP# }

X { PHONE#, OFF# }
  PRIMARY KEY { PHONE# }

Y { PROJ#, DEPT# }
  PRIMARY KEY { PROJ# }

Z { OFF#, DEPT# }
  PRIMARY KEY { OFF# }
```

However, X is PHONE2, Y is a projection of PROJ2, and Z is a projection of OFFICE2. Hence our collection of 3NF relvars is simply

```
DEPT3 { DEPT#, DBUDGET, MGR_EMP# }
      PRIMARY KEY { DEPT# }
      ALTERNATE KEY { MGR_EMP# }

EMP3 { EMP#, PROJ#, PHONE# }
     PRIMARY KEY { EMP# }

SALHIST3 { EMP#, DATE, JOBTITLE, SALARY }
         PRIMARY KEY { EMP#, DATE }

PROJ3 { PROJ#, DEPT#, PBUDGET }
      PRIMARY KEY { PROJ# }

OFFICE3 { OFF#, DEPT#, AREA }
        PRIMARY KEY { OFF# }

PHONE3 { PHONE#, OFF# }
       PRIMARY KEY { PHONE# }
```

Finally, it's easy to see that each of these 3NF relvars is in fact in BCNF.

Note that, given certain (reasonable) additional semantic constraints, this collection of BCNF relvars is **strongly redundant** [6.1], in that the projection of relvar PROJ3 over {PROJ#,DEPT#}

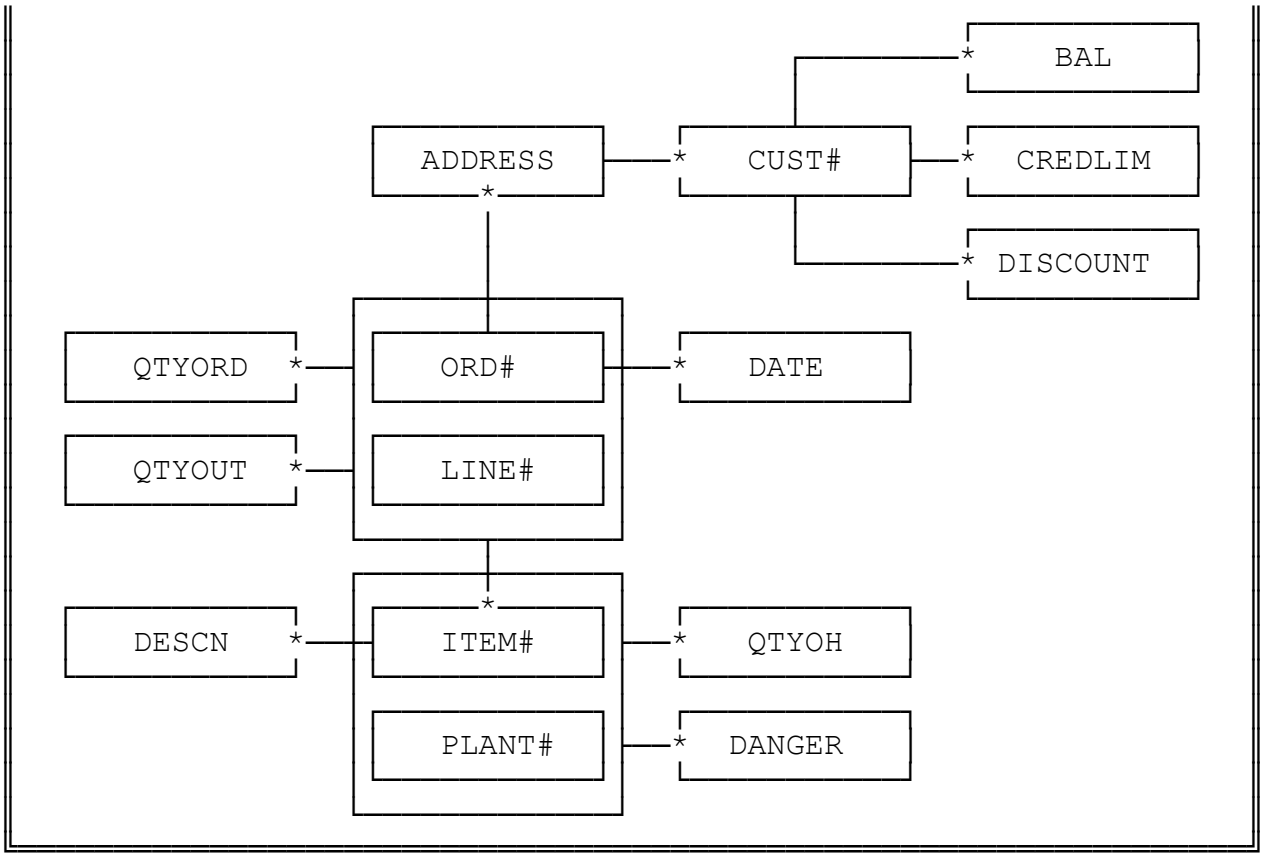
is at all times equal to a projection of the join of EMP3 and PHONE3 and OFFICE3.

Observe finally that it's possible to "spot" the BCNF relvars from the FD diagram (how?). *Answer:* Loosely, there'll be one such relvar for each box that has an arrow emerging from it; that relvar will include the attributes from that original box as a candidate key, together with an attribute for every box pointed to from the original box (and no other attributes). Of course, some refinement is needed to this loose statement in order to take care of relvars like DEPT3 that have two or more candidate keys. *Note:* We don't claim that it's *always* possible to "spot" a BCNF decomposition——only that it's often possible to do so in practical cases.

To revert to the company database example: As a subsidiary exercise——not much to do with normalization as such, but very relevant to database design in general——try extending the foregoing design to incorporate the necessary *foreign key* specifications as well. *Answer:*

```
DEPT3 { DEPT#, DBUDGET, MGR_EMP# }
      PRIMARY KEY { DEPT# }
      ALTERNATE KEY { MGR_EMP# }
      FOREIGN KEY { RENAME MGR_EMP# AS EMP# } REFERENCES EMP3

EMP3 { EMP#, PROJ#, PHONE# }
     PRIMARY KEY { EMP# }
     FOREIGN KEY { PROJ# } REFERENCES PROJ3
     FOREIGN KEY { PHONE# } REFERENCES PHONE3

SALHIST3 { EMP#, DATE, JOBTITLE, SALARY }
         PRIMARY KEY { EMP#, DATE }
         FOREIGN KEY { EMP# } REFERENCES EMP3

PROJ3 { PROJ#, DEPT#, PBUDGET }
      PRIMARY KEY { PROJ# }
      FOREIGN KEY { DEPT# } REFERENCES DEPT3

OFFICE3 { OFF#, DEPT#, AREA }
        PRIMARY KEY { OFF# }
        FOREIGN KEY { DEPT# } REFERENCES DEPT3

PHONE3 { PHONE#, OFF# }
       PRIMARY KEY { PHONE# }
       FOREIGN KEY { OFF# } REFERENCES OFFICE3
```

12.4 The figure below shows the most important FDs for this exercise. The semantic assumptions are as follows:



- No two customers have the same ship-to address.
- Each order is identified by a unique order number.
- Each detail line within an order is identified by a line number, unique within the order.

An appropriate set of BCNF relvars is as follows:

```
CUST { CUST#, BAL, CREDLIM, DISCOUNT }
     KEY { CUST# }

SHIPTO { ADDRESS, CUST# }
       KEY { ADDRESS }

ORDHEAD { ORD#, ADDRESS, DATE }
        KEY { ORD# }

ORDLINE { ORD#, LINE#, ITEM#, QTYORD, QTYOUT }
        KEY { ORD#, LINE# }

ITEM { ITEM#, DESCN }
     KEY { ITEM# }
```

Copyright (c) 2003 C. J. Date

```
IP { ITEM#, PLANT#, QTYOH, DANGER }
   KEY { ITEM#, PLANT# }
```

12.5 Consider the processing that must be performed by a program handling orders. We assume that the input order specifies customer number, ship-to address, and details of the items ordered (item numbers and quantities).

```
retrieve CUST WHERE CUST# = input CUST# ;
check balance, credit limit, etc. ;
retrieve SHIPTO WHERE ADDRESS = input ADDRESS
                AND   CUST#   = input CUST#
/* this checks the ship-to address */ ;
IF everything is OK THEN process the order ; END IF ;
```

If 99 percent of customers actually have only one ship-to address, it would be rather inefficient to put that address in a relvar other than CUST (if we consider only that 99 percent, ADDRESS is in fact functionally dependent on CUST#). We can improve matters as follows. For each customer we designate one valid ship-to address as that customer's *primary* address. For the 99 percent, of course, the primary address is the only address. Any other addresses we refer to as *secondary*. Relvar CUST can then be redefined as

```
CUST { CUST#, ADDRESS, BAL, CREDLIM, DISCOUNT }
     KEY { CUST# }
```

and relvar SHIPTO can be replaced by

```
SECOND { ADDRESS, CUST# }
       KEY { ADDRESS }
```

Here CUST contains the primary address, and SECOND contains all secondary addresses (and corresponding customer numbers). These relvars are both in BCNF. The order processing program now looks like this:

```
retrieve CUST WHERE CUST# = input CUST# ;
check balance, credit limit, etc. ;
IF retrieved ADDRESS =/ input ADDRESS THEN
   retrieve SECOND WHERE ADDRESS = input ADDRESS
                   AND   CUST#   = input CUST#
   /* this checks the ship-to address */ ;
END IF :
IF everything is OK THEN process the order ; END IF ;
```

The advantages of this approach include the following:

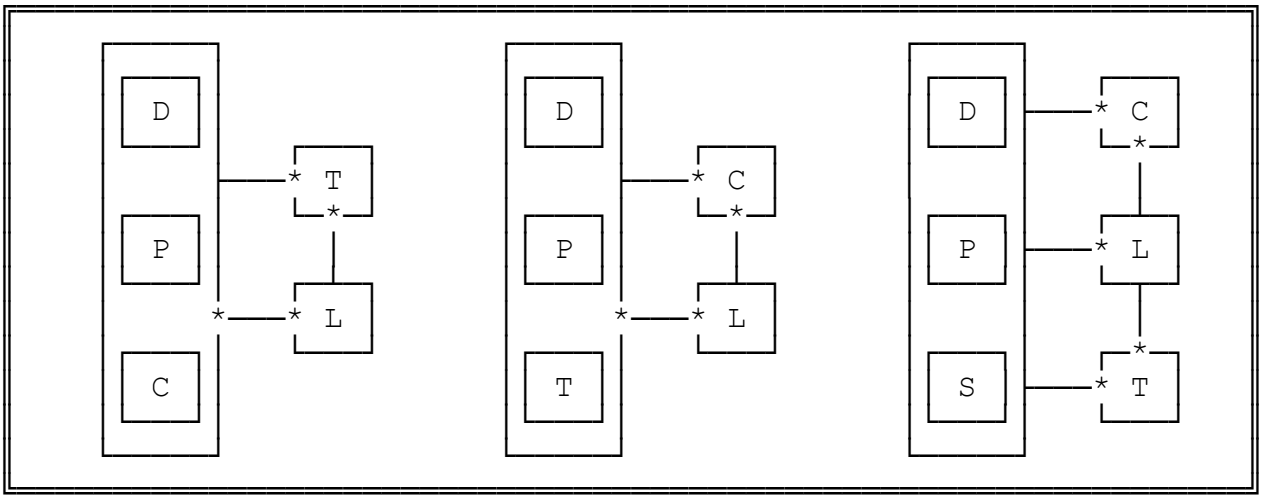
- Processing is simpler (and possibly more efficient) for 99 percent of customers.

- If the ship-to address is omitted from the input order, the primary address could be used by default.

- Suppose the customer can have a different discount for each ship-to address. With the original approach (shown as the answer to the previous exercise), the DISCOUNT attribute would have to be moved to the SHIPTO relvar, making processing still more complicated. With the revised approach, however, the primary discount (corresponding to the primary address) can be represented by an appearance of DISCOUNT in CUST, and secondary discounts by a corresponding appearance of DISCOUNT in SECOND. Both relvars are still in BCNF, and processing is again simpler for 99 percent of customers.

To sum up: Isolating exceptional cases seems to be a valuable technique for obtaining the best of both worlds——i.e., combining the advantages of BCNF with the simplification in retrieval that can occur if the restrictions of BCNF are violated.

12.6 The figure below shows the most important FDs. A possible collection of relvars is:




```
SCHED { L, T, C, D, P }
      KEY { L }
      KEY { T, D, P }
      KEY { C, D, P }

STUDY { S, L }
      KEY { S, L }
```

12.7 NADDR is in 2NF but not 3NF (and hence not BCNF). A better design might be:

```
NSZ { NAME, STREET, ZIP }
    KEY { NAME }

ZCS { ZIP, CITY, STATE }
    KEY { ZIP }
```

These two relvars are both in BCNF. Note, however, that:

- Since STREET, CITY, and STATE are almost invariably required together (think of printing a mailing list), and since zip codes don't change very often, it might be argued that such a decomposition is hardly worthwhile. (In other words, normalization should generally be carried out with respect to all *relevant* dependencies——not necessarily all dependencies that exist.)

- Observe in particular that retrieving the full address for a given NAME now requires a join (although that join could be concealed from the user by defining NADDR as a view of NSZ and ZCS). Hence, it might be argued that normalization to BCNF is *good for update but bad for retrieval*——i.e., the redundancy that occurs in the absence of full normalization certainly causes problems with update but might help with retrieval.* Redundancy causes difficulties if it's *uncontrolled;* but *controlled* redundancy (i.e., redundancy that's declared to the DBMS, and managed by the DBMS) might be acceptable in some situations. (Note, by the way, that the redundancy we're talking about here is redundancy *at the logical level*——i.e., it's visible to the user.)

———————

* On the other hand, such redundancy can actually hinder certain retrievals (i.e., it can make the corresponding queries more awkward to formulate), as we'll see in Section 13.5 in the next chapter.

———————

- The FD { STREET, CITY, STATE } → ZIP isn't directly represented by this design; instead, it'll have to be maintained separately, either declaratively (if the DBMS supports a declarative integrity language along the lines of the one sketched in Chapter 9), or procedurally otherwise. In

fact, of course, relvars NSZ and ZCS are not *independent* in Rissanen's sense [12.6].

Note: We saw in the answer to Exercise 11.15 that in fact the FD ZIP → { CITY, STATE } does *not* hold in practice. As a subsidiary exercise, therefore, revise your answer to Exercise 12.7 to take this fact into account. *Answer:* We don't give a full answer here, but remark that the techniques illustrated in the answer to Exercise 12.5 are relevant.

12.8 This one is surprisingly tricky!—but the following should suffice: *Let spqt be an arbitrary tuple appearing in relvar SPQ, and let s and pq be the S# and PQ values, respectively, appearing in that tuple spqt. Let pqt be an arbitrary tuple appearing in pq, and let p and q be the P# and QTY values, respectively, appearing in that tuple pqt. Then (a) s doesn't appear in any tuple of SPQ apart from spqt; (b) p doesn't appear in any tuple of pq apart from pqt; (c) s supplies p in quantity q; (d) s doesn't supply any other parts.*

***** End of Chapter 12 *****

Chapter 13

Further Normalization II: Higher Normal Forms

Principal Sections

- MVDs and 4NF
- JDs and 5NF
- The normalization procedure summarized
- A note on denormalization
- Orthogonal design (a digression)
- Other normal forms

General Remarks

This chapter can just be skimmed, or even skipped entirely, if desired. Obviously, the following notes assume the chapter is *not* skipped.

The basic idea, of course, is that FDs can be generalized and those generalizations can then be used to help avoid further update anomalies. Perhaps a better way to put it is this: FDs are just a pragmatically important special case of what's called a *JD;* JDs are the general case (of this particular kind of integrity constraint). And MVDs are a kind of halfway house between FDs and JDs, *very* loosely speaking.

It's important to understand that (as in the previous chapter) we're still talking about *nonloss decomposition,* with projection as the decomposition operator and join as the recomposition operator.

13.2 MVDs and 4NF

Mostly self-explanatory. Possible points for the instructor to note:

Copyright (c) 2003 C. J. Date

- If $A \rightarrow B$, then certainly $A \rightarrow\!\!\!\!\rightarrow B$, but the converse is not true. (If $A \rightarrow\!\!\!\!\rightarrow B$ **and the cardinality of the set of *B* values matching any given *A* value is always one,** then $A \rightarrow B$.)

- An MVD $A \rightarrow\!\!\!\!\rightarrow B$ is **trivial** if either A is a superset of B or the union of A and B is the entire heading.

- Let $R\{A,B,C\}$ be a relvar, where A, B, and C are sets of attributes. Then R is equal to the join of its projections on $\{A,B\}$ and $\{A,C\}$ iff* R satisfies the MVDs $A \rightarrow\!\!\!\!\rightarrow B \mid C$. (This theorem, due to Fagin, is a stronger version of Heath's theorem as defined in Chapter 12.)

---

* Recall from Chapter 7 that "iff" stands for "if and only if."

- "Fourth" normal form is really *fifth* if you count——it was called 4NF because BCNF was still being called *third* at the time (at least by some people).

- 4NF is always achievable.

- In practice, the first step in eliminating RVAs should be to separate them out into separate relvars. E.g., starting with HCTX, split into HCT { COURSE, TEACHERS } and HCX { COURSE, TEXTS }; then ungrouping HCT and HCX will take us straight to CT and CX. (See the discussion of Answer 12.3 in the previous chapter.)

- In practice, if a relvar is in BCNF, it's almost certainly in 4NF too (one reason why the chapter can probably be skipped, especially if the emphasis is on practice rather than theory).

If you want to get into more formalism, see the algorithm in Answer 13.4 for obtaining 4NF.

The section includes the following as an inline exercise: Give a relational expression by which CTX can be derived from HCTX. *Answer:*

```
( HCTX UNGROUP TEACHERS ) UNGROUP TEXTS
```

Of course, the following works too:

```
( HCTX UNGROUP TEXTS ) UNGROUP TEACHERS
```

In other words, the ungroupings can be done in either order.

13.3 JDs and 5NF

Again mostly self-explanatory. Possible points for the instructor to note:

- The "cyclic constraint" stuff (this is a helpful intuitive characterization of a relvar that's in 4NF but not 5NF; perhaps mention that such constraints seem to be rare in practice?——see the seventh bullet in this list).

- *A* →→ *B* | *C* ≡ * { *AB*, *AC* }. In other words, MVDs are indeed just a special case of JDs——and if JDs had been defined first, there wouldn't have been any need to define MVDs, as such, at all. However, MVDs (like FDs) do have an intuitive interpretation that's not too hard to understand, whereas JDs in their full generality don't seem to (the best I can come up with is the "cyclic constraint" stuff).

- JDs are **the most general form of dependency possible** (using the term "dependency" in a very special sense!), and 5NF (*aka* PJ/NF) is **the final normal form** with respect to projection and join. *Note:* "Projection and join" here refers (of course) to those operators as classically understood. In Chapter 23, we'll be defining generalized versions of those operators; then we'll generalize JDs as well, and come up with a *sixth* normal form that is qualitatively different from (and "more normalized" than) 5NF. But we're not going to get into those generalizations in this chapter; for now, 5NF *is* the "final" normal form.

- A JD *{*A*,*B*,...,*Z*} is **trivial** iff at least one of *A*, *B*, ..., *Z* is the **identity projection**.

- A JD *{*A*,*B*,...,*Z*} is **implied by candidate keys** iff each of A, B, ..., Z is in fact a **superkey**.

- 5NF is always achievable.

- In practice, if a relvar is in 4NF, it's almost certainly in 5NF too. (Personally, I've only ever seen two genuine relvars——i.e., actual relvars in actual business databases——that were in 4NF and not 5NF.)

- A nice theorem [13.11] that seems not to as widely known as it might be: *If a relvar is in 3NF and has no composite keys, it's in 5NF.* This result doesn't mean that, as a database designer, you don't have to know about MVDs and JDs and 4NF and 5NF*——but it does mean there are many situations where you don't have to worry about them (because you have a very simple test, a test that will often be satisfied in practice, for checking whether a given relvar is in fact in the final normal form). *Note:* Reference [13.11] gives Ron Fagin and myself as the source of this theorem. While I was the one who conjectured that it might be true, Ron was really the one who did the work of proving it to be so (not a very equal division of labor!).

———————

* It also doesn't mean we can force a non5NF relvar into 5NF by simply introducing a noncomposite surrogate key! Introducing a new key doesn't mean previously existing keys, composite or otherwise, suddenly aren't keys after all. (Apologies for what might look like an extremely obvious remark, but people often seem to misconstrue the theorem for some reason.)

———————

Regarding the SPJ example, the text says this: "Observe that the result of the first join is to produce a copy of the original SPJ relation plus one additional (spurious) tuple, and the effect of the second join is then to eliminate that spurious tuple, thereby bringing us back to the original SPJ relation. In other words, the original SPJ relation is 3-decomposable. *Note:* The net result is the same whatever pair of projections we choose for the first join, though the intermediate result is different in each case. *Exercise:* Check this claim."

Answer:

- SP JOIN PJ yields the spurious tuple (S2,P1,J2); there's no (J2,S2) tuple in JS; hence the final result is SPJ (as we've already seen).

- PJ JOIN JS yields the spurious tuple (S2,P2,J1); there's no (S2,P2) tuple in SP; hence the final result is SPJ again.

- JS JOIN SP yields the spurious tuple (S1,P2,J2); there's no (P2,J2) tuple in PJ; hence the final result is SPJ once again.

The section also includes the following: "We've seen ... that relvar SPJ, with its JD *{SP,PJ,JS}, can be 3-decomposed. The question is, *should* it be? And the answer is probably *yes*. Relvar SPJ (with its JD) suffers from a number of problems over update operations, problems that are removed when it is 3-decomposed. Some examples of such problems are illustrated in Fig. 13.5. Consideration of what happens after 3-decomposition is left as an exercise." *No answer provided.*

"The problem with relvar SPJ is that it involves a JD that's not an MVD, and hence not an FD either. *Exercise: Why* is this a problem, exactly?" *Answer:* Because it means (among other things) that procedural code is needed, in general, in order to avoid certain update anomalies——anomalies that could be avoided declaratively (via the system's key uniqueness enforcement mechanism) if the relvar were in 5NF.

13.4 The Normalization Procedure Summarized

Again fairly self-explanatory. The "attractive parallelism" mentioned in the text is worth highlighting:

- *R* is in BCNF iff every FD satisfied by *R* is implied by the candidate keys of *R*.

- *R* is in 4NF iff every MVD satisfied by *R* is implied by the candidate keys of *R*.

- *R* is in 5NF iff every JD satisfied by *R* is implied by the candidate keys of *R*.

Also stress that normalization is not a panacea [13.9].

The following list of normalization principles (taken from reference [13.10]) is probably worth a brief review:

1. A non5NF relvar should be decomposed into a set of 5NF projections. *(Even if you experience performance problems, owing to product deficiencies, you should denormalize only as a last resort.)*

2. The original relvar should be reconstructable by joining the projections back together again. *(The decomposition must be nonloss.)*

3. The decomposition process should preserve dependencies. *(Preferably decompose into independent projections——though as we know, this objective and the objective of decomposing to 5NF, or even just to BCNF, can unfortunately be in conflict.)*

4. Every projection should be needed in the reconstruction process. *(This one is often overlooked!——or at least taken for granted. But it's worth spelling out explicitly.)*

5. Stop normalizing as soon as all relvars are in 5NF. *(This one isn't as firm as the first four. But the general idea is not to "normalize too much." Though I should mention that, when we get to Chapter 23, we'll find cases where we really ought to normalize "as far as possible.")*

13.5 A Note on Denormalization

It's interesting that we hear talk of "denormalization" in the commercial world all the time, and yet textbooks (including earlier editions of this one) typically don't discuss it at all, or define it, or even mention it! This observation is the justification for including such a discussion here.

Stress the point that denormalization (as the term is usually understood) is an issue in current products only because those products don't adequately distinguish between the logical and physical levels of the system (base relvars are, typically, physically stored in those products). Forward pointer to Appendix A?

Also stress the point that (contrary to conventional wisdom) denormalization can be bad for retrieval as well as for update.* In fact, denormalization flies in the face of the objective of application-independent design: It "optimizes" the design (maybe) for some applications at the expense of others. *Normalization,* by contrast, is more application-neutral.

———————

* Bad, that is, both physically and logically——physically because (fairly obviously) it can make some queries perform worse; logically because (less obviously) it can make some queries harder to formulate (e.g., suppose relvar S satisfies the FD CITY → STATUS, and consider the query "Get average status per city").

———————

See also the annotation to reference [13.6].

13.6 Orthogonal Design (a digression)

A little more science! *The Principle of Orthogonal Design:* Let *A* and *B* be any two base relvars* in the database. Then there must not exist nonloss decompositions of *A* and *B* into *A1, ..., Am* and *B1, ..., Bn* (respectively) such that some projection *Ai* in the set *A1, ..., Am* and some projection *Bj* in the set *B1, ..., Bn* have overlapping meanings. (This version of the principle subsumes the simpler version, because one nonloss decomposition that always exists for relvar *R* is the identity projection of *R,* i.e., the projection of *R* over all of its attributes.)

_____

* Recall that, from the user's point of view, *all* relvars are base ones (apart from views defined as mere shorthands); i.e., the principle applies to the design of all "expressible" databases, not just to the "real" database——*The Principle of Database Relativity* at work once again. Of course, analogous remarks apply to the principles of normalization also.

_____

It's predicates, not names, that represent data semantics.

Mention "orthogonal decomposition" (this will be relevant when we get to distributed databases in Chapter 21).

Violating *The Principle of Orthogonal Design* in fact violates *The Information Principle!* The principle is just formalized common sense, of course (like the principles of further normalization). Remind students of the relevance of the principle to updating union, intersection, and difference views (Chapter 10).

13.7 Other Normal Forms

You're welcome to skip this section. If you do cover it, note that there's some confusion in the literature over exactly what DK/NF is (see, e.g., "The Road to Normalization," by Douglas W. Hubbard and Joe Celko, *DBMS,* April 1994). *Note:* After I first wrote these notes, the topic of DK/NF came up on the website *www.dbdebunk.com*. I've attached my response to that question as an appendix to this chapter of the manual.

References and Bibliography

Reference [13.15] is a classic and should be distributed to students if at all possible.

The annotation to reference [13.14] says this: "The two embedded MVDs [in relvar CTXD] would have to be stated as additional, explicit constraints on the relvar. The details are left as an exercise." *Answer:*

```
CONSTRAINT EMVD_ON_CTXD
   CTXD { COURSE, TEACHER, TEXT } =
   CTXD { COURSE, TEACHER } JOIN CTXD { COURSE, TEXT } ;
```

Note that this constraint is *much* harder to state in SQL, because SQL doesn't support relational comparisons! Here it is in SQL:

```
CREATE ASSERTION EMVD_ON_CTXD
( NOT EXISTS ( SELECT DISTINCT COURSE, TEACHER, TEXT
               FROM   CTXD AS CTXD1
               WHERE  NOT EXISTS
                    ( SELECT DISTINCT COURSE, TEACHER, TEXT
                      FROM ( ( SELECT DISTINCT COURSE, TEACHER
                               FROM   CTXD ) AS POINTLESS1
                               NATURAL JOIN
                             ( SELECT DISTINCT COURSE, TEXT
                               FROM   CTXD ) AS POINTLESS2 ) )
                             AS CTXD2
                      WHERE  CTXD1.COURSE  = CTXD2.COURSE
                      AND    CTXD1.TEACHER = CTXD2.TEACHER
                      AND    CTXD1.TEXT    = CTXD2.TEXT )
AND
( NOT EXISTS ( SELECT DISTINCT COURSE, TEACHER, TEXT
               FROM ( ( SELECT DISTINCT COURSE, TEACHER
                        FROM   CTXD ) AS POINTLESS1
                       NATURAL JOIN
                      ( SELECT DISTINCT COURSE, TEXT
                        FROM   CTXD ) AS POINTLESS2 ) )
                         AS CTXD2
               WHERE  NOT EXISTS
                    ( SELECT DISTINCT COURSE, TEACHER, TEXT
                      FROM   CTXD AS CTXD1
                      WHERE  CTXD1.COURSE  = CTXD2.COURSE
                      AND    CTXD1.TEACHER = CTXD2.TEACHER
                      AND    CTXD1.TEXT    = CTXD2.TEXT ) ;
```

You might want to discuss this SQL formulation in detail.

Answers to Exercises

13.1 Here first is the MVD for relvar CTX (algebraic version):

```
    CONSTRAINT CTX_MVD CTX = CTX { COURSE, TEACHER } JOIN
                             CTX { COURSE, TEXT } ;
```

Calculus version:

```
    CONSTRAINT CTX_MVD CTX =
             { CTXX.COURSE, CTXX.TEACHER, CTXY.TEXT }
                            WHERE CTXX.COURSE = CTXY.COURSE ;
```

CTXX and CTXY are range variables ranging over CTX.

Second, here is the JD for relvar SPJ (algebraic version):

```
    CONSTRAINT SPJ_JD SPJ = SPJ { S#, P# } JOIN
                            SPJ { P#, J# } JOIN
                            SPJ { J#, S# } ;
```

Calculus version:

```
    CONSTRAINT SPJ_JD SPJ =
             { SPJX.S#, SPJY.P#, SPJZ.J# } WHERE SPJX.P# = SPJY.P#
                                            AND SPJY.J# = SPJZ.J#
                                            AND SPJZ.S# = SPJX.S# ;
```

SPJX, SPJY, and SPJZ are range variables ranging over SPJ.

13.2 Note first that *R* contains every *a* value paired with every *b* value, and further that the set of all *a* values in *R*, *S* say, is the same as the set of all *b* values in *R*. Loosely speaking, therefore, the body of *R* is equal to the Cartesian product of set *S* with itself; more precisely, *R* is equal to the Cartesian product of its projections *R*{*A*} and *R*{*B*}. *R* thus satisfies the following MVDs (which are *not* trivial, please note, since they're certainly not satisfied by all binary relvars):

$$\{\ \} \twoheadrightarrow A \mid B$$

Equivalently, *R* satisfies the JD *{*A*,*B*} (remember that join degenerates to Cartesian product when there are no common attributes). It follows that *R* isn't in 4NF, and it can be nonloss-decomposed into its projections on *A* and *B*.* *R* is, however, in BCNF (it's all key), and it satisfies no nontrivial FDs.

---

* Those projections will have identical bodies, of course. For that reason, it might be better to define just one of them as a

base relvar, and define *R* as a view over that base relvar (the Cartesian product of that base relvar with itself, loosely speaking).

Note: *R* also satisfies the MVDs

A →→ *B* | { }

and

B →→ *A* | { }

However, these MVDs *are* trivial, since they're satisfied by every binary relvar *R* with attributes *A* and *B*.

13.3 First we introduce three relvars

```
REP     { REP#, ... }
        KEY { REP# }

AREA    { AREA#, ... }
        KEY { AREA# }

PRODUCT { PROD#, ... }
        KEY { PROD# }
```

with the obvious interpretation. Second, we can represent the relationship between sales representatives and sales areas by a relvar

```
RA { REP#, AREA# }
   KEY { REP#, AREA# }
```

and the relationship between sales representatives and products by a relvar

```
RP { REP#, PROD# }
   KEY { REP#, PROD# }
```

(both of these relationships are many-to-many).

Next, we're told that every product is sold in every area. So if we introduce a relvar

```
AP { AREA#, PROD# }
   KEY { AREA#, PROD# }
```

Copyright (c) 2003 C. J. Date

to represent the relationship between areas and products, then we have the constraint (let's call it C) that

```
AP = AREA { AREA# } JOIN PRODUCT { PROD# }
```

Notice that constraint C implies that relvar AP isn't in 4NF (see Exercise 13.2). In fact, relvar AP doesn't give us any information that can't be obtained from the other relvars; to be precise, we have

```
AP { AREA# } = AREA { AREA# }
```

and

```
AP { PROD# } = PRODUCT { PROD# }
```

But let's assume for the moment that relvar AP *is* included in our design anyway.

No two representatives sell the same product in the same area. In other words, given an {AREA#,PROD#} combination, there's exactly one responsible sales representative (REP#), so we can introduce a relvar

```
APR { AREA#, PROD#, REP# }
    KEY { AREA#, PROD# }
```

in which (to make the FD explicit)

```
{ AREA#, PROD# } → REP#
```

(of course, specification of the combination {AREA#,PROD#} as a key is sufficient to express this FD). Now, however, relvars RA, RP, and AP are all redundant, since they're all projections of APR; they can therefore all be dropped. In place of constraint C, we now need constraint C1:

```
APR { AREA#, PROD# } = AREA { AREA# } JOIN PRODUCT { PROD# }
```

This constraint must be stated separately and explicitly (it isn't "implied by keys").

Also, since every representative sells all of that representative's products in all of that representative's areas, we have the additional constraint C2 on relvar APR:

```
REP# →→ AREA# | PROD#
```

(a nontrivial MVD; relvar APR isn't in 4NF). Again the constraint must be stated separately and explicitly.

Thus the final design consists of the relvars REP, AREA, PRODUCT, and APR, together with the constraints C1 and C2:

```
CONSTRAINT C1 APR { AREA#, PROD# } =
              AREA { AREA# } JOIN PRODUCT { PROD# } ;

CONSTRAINT C2 APR =
              APR { REP#, AREA# } JOIN APR { REP#, PROD# } ;
```

This exercise illustrates very clearly the point that, in general, the normalization discipline is adequate to represent *some* semantic aspects of a given problem (basically, dependencies that are implied by keys, where by "dependencies" we mean FDs, MVDs, or JDs), but explicit statement of additional dependencies might also be needed for other aspects, and some aspects can't be represented in terms of such dependencies at all. It also illustrates the point (once again) that it isn't always desirable to normalize "all the way" (relvar APR is in BCNF but not in 4NF).

Note: As a subsidiary exercise, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration. Might such a design mean that some of the comments in the previous paragraph no longer apply?

13.4 The revision is straightforward——all that's necessary is to replace the references to FDs and BCNF by analogous references to MVDs and 4NF, thus:

1. Initialize *D* to contain just *R*.

2. For each non4NF relvar *T* in *D*, execute Steps 3 and 4.

3. Let $X \twoheadrightarrow Y$ be an MVD for *T* that violates the requirements for 4NF.

4. Replace *T* in *D* by two of its projections, that over *X* and *Y* and that over all attributes except those in *Y*.

13.5 This is a "cyclic constraint" example. The following design is suitable:

```
REP     { REP#, ... }
        KEY { REP# }

AREA    { AREA#, ... }
        KEY { AREA# }

PRODUCT { PROD#, ... }
        KEY { PROD# }
```

Copyright (c) 2003 C. J. Date

```
RA { REP#, AREA# }
   KEY { REP#, AREA# }

AP { AREA#, PROD# }
   KEY { AREA#, PROD# }

PR { PROD#, REP# }
   KEY { PROD#, REP# }
```

Also, the user needs to be informed that the join of RA, AP, and PR does *not* involve any "connection trap":

```
CONSTRAINT NO_TRAP
  ( RA JOIN AP JOIN PR ) { REP#, AREA# }  = RA AND
  ( RA JOIN AP JOIN PR ) { AREA#, PROD# } = AP AND
  ( RA JOIN AP JOIN PR ) { PROD#, REP# }  = PR ;
```

Note: As with Exercise 13.3, you might like to consider whether a design involving RVAs might be appropriate for the problem under consideration.

13.6 Perhaps surprisingly, the design *does* conform to normalization principles! First, SX and SY are both in 5NF. Second, the original suppliers relvar can be reconstructed by joining SX and SY back together. Third, neither SX nor SY is redundant in that reconstruction process. Fourth, SX and SY are independent in Rissanen's sense.

Despite the foregoing observations, the design is very bad, of course; to be specific, it involves some obviously undesirable redundancy. But the design isn't bad because it violates the principles of normalization; rather, it's bad because it violates *The Principle of Orthogonal Design,* as explained in Section 13.6. Thus, we see that following the principles of normalization are *necessary but not sufficient* to ensure a good design. We also see that (as stated in Section 13.6) the principles of normalization and *The Principle of Orthogonal Design* complement each other, in a sense.

Appendix (DK/NF)

This appendix consists (apart from this introductory paragraph) of the text——slightly edited here——of a message posted on the website *www.dbdebunk.com* in May 2003. It's my response to a question from someone I'll refer to here as Victor.

(Begin quote)

Victor has "trouble understanding ... domain-key normal form (DK/NF)." I don't blame him; there's certainly been some serious nonsense published on this topic in the trade press and elsewhere. Let me see if I can clarify matters.

DK/NF is best thought of as a straw man (sorry, straw person). It was introduced by Ron Fagin in his paper "A Normal Form for Relational Databases that Is Based on Domains and Keys," *ACM TODS 6,* No. 3 (September 1981). As Victor says (more or less), Fagin defines a relvar *R* to be in DK/NF if and only if every constraint on *R* is a logical consequence of what he (Fagin) calls the *domain constraints* and *key constraints* on *R*. Here:

- A domain constraint——better called an *attribute* constraint——is simply a constraint to the effect a given attribute *A* of *R* takes its values from some given domain *D*.

- A key constraint is simply a constraint to the effect that a given set *A, B, ..., C* of *R* constitutes a key for *R*.

Thus, **if** *R* is in DK/NF, **then** it is sufficient to enforce the domain and key constraints for *R,* and *all* constraints on *R* will be enforced automatically. And enforcing those domain and key constraints is, of course, very simple (most DBMS products do it already). To be specific, enforcing domain constraints just means checking that attribute values are always values from the applicable domain (i.e., values of the right type); enforcing key constraints just means checking that key values are unique.

The trouble is, lots of relvars aren't in DK/NF in the first place. For example, suppose there's a constraint on *R* to the effect that *R* must contain at least ten tuples. Then that constraint is certainly not a consequence of the domain and key constraints that apply to *R,* and so *R* isn't in DK/NF. The sad fact is, not all relvars can be reduced to DK/NF; nor do we know the answer to the question "Exactly when *can* a relvar be so reduced?"

Now, it's true that Fagin proves in his paper that **if** relvar *R* is in DK/NF, **then** *R* is automatically in 5NF (and hence 4NF, BCNF, etc.) as well. However, it's wrong to think of DK/NF as another step in the progression from 1NF to 2NF to ... to 5NF, because 5NF is always achievable, but DK/NF is not.

It's also wrong to say there are "no normal forms higher than DK/NF." In recent work of my own——documented in the book *Temporal Data and the Relational Model,* by myself with Hugh Darwen and Nikos Lorentzos (Morgan Kaufmann, 2003)——my coworkers and I have come up with a new *sixth* normal form, 6NF. 6NF is higher than 5NF (all 6NF relvars are in 5NF, but the converse isn't true);

moreover, 6NF is always achievable, but it isn't implied by DK/NF. In other words, there are relvars in DK/NF that aren't in 6NF. A trivial example is:

```
EMP { EMP#, DEPT#, SALARY } KEY { EMP# }
```

(with the obvious semantics).

Victor also asks: "If a [relvar] has an atomic primary key and is in 3NF, is it automatically in DK/NF?" **No**. If the EMP relvar just shown is subject to the constraint that there must be at least ten employees, then EMP is in 3NF (and in fact 5NF) but not DK/NF. (Incidentally, this example also answers another of Victor's questions: "Can [we] give "an example of a [relvar] that's in 5NF but not ... in DK/NF?") *Note:* I'm assuming here that the term "atomic key" means what would more correctly be called a *simple* key (meaning it doesn't involve more than one attribute). I'm also assuming that the relvar in question has *just one* key, which we might harmlessly regard as the "primary" key. If either of these assumptions is invalid, the answer to the original question is probably "no" even more strongly!

The net of all of the above is that DK/NF is (at least at the time of writing) a concept that's of some considerable theoretical interest but not yet of much practical ditto. The reason is that, while it would be nice if all relvars in the database were in DK/NF, we know that goal is impossible to achieve in general, nor do we know when it *is* possible. For practical purposes, stick to 5NF (and 6NF). Hope this helps!

(End quote)

***** End of Chapter 13 *****

Copyright (c) 2003 C. J. Date

Chapter 14

S e m a n t i c M o d e l i n g

Principal Sections

- The overall approach
- The E/R model
- E/R diagrams
- DB design with the E/R model
- A brief analysis

General Remarks

The field of "semantic modeling" encompasses more than just database design, but for obvious reasons the emphasis in this chapter is on database design aspects (though the first two sections do consider the wider perspective briefly, and so does the annotation to several of the references at the end of the chapter). The chapter shouldn't be skipped, but portions of it might be skipped. You could also beef up the treatment of "E/R modeling" if you like.

Let me repeat the following remarks from the preface to this manual:

> You could also read Chapter 14 earlier if you like, possibly right after Chapter 4. Many instructors like to treat the entity/relationship material much earlier than I do. For that reason I've tried to make Chapter 14 more or less self-contained, so that it can be read "early" if you like.

And the expanded version of these remarks from the preface to the book itself:

> Some reviewers of earlier editions complained that database design issues were treated too late. But it's my feeling that students aren't ready to design databases properly or to appreciate design issues fully until they have some understanding of what databases are and how they're used; in other words, I believe it's important to spend some time on the relational model and related matters before exposing the student to design questions. Thus, I still believe Part III is in the right place. (That said, I do recognize that many instructors prefer to treat the entity/relationship material much earlier. To that end, I've tried to make Chapter 14 more

Copyright (c) 2003 C. J. Date

or less self-contained, so that they can bring it in immediately after, say, Chapter 4.)

On to the substance. The predicate stuff is important yet again. Indeed, my own preferred way of doing database design is to start by **writing down the predicates** (i.e., the *external* predicates, *aka* the "business rules"). However, most people prefer to draw pictures. Pictures can certainly be helpful, but they don't even begin to capture enough of the semantics to do the whole job. In this connection, note the following remarks from the annotation to reference [14.39]:

> E/R diagrams and similar formalisms ... are strictly less powerful than formal logic ... [They] are completely incapable of dealing with ... anything involving explicit quantification, which includes almost all integrity constraints ... (The quantifiers were invented by Frege in 1879, which makes E/R diagrams "a pre-1879 kind of logic"!)

14.2 The Overall Approach

Summarize the four steps. The first step is informal, the others formal. Stress the point that the rules and operators are just as much part of the model as the objects are; the operators might be thought by some people to be less important than the objects and rules from a database design point of view, but we need the operators to express the rules! (And, to say it one more time, the rules are *crucial*.)

Note: The section uses RM/T to illustrate the ideas, but you can certainly use something else instead if you don't care for RM/T for some reason. I prefer RM/T myself because——unlike most other approaches——it very explicitly addresses the rules and the operators as well as the objects.* (I also like RM/T's categorization of entities into kernel, characteristic, and associative entities, though that categorization isn't discussed in the body of the chapter. See the annotation to reference [14.7].)

———————

* That said, I should say too that a lot more work is needed on these aspects of the model.

———————

Important: The very same object in the real world might legitimately be regarded as an entity by some people and as a

relationship by others.* It follows that, while distinguishing between entities and relationships can be useful, *intuitively,* it's not a good idea to make that distinction *formal.* This criticism applies to "the E/R model"; it also applies (applied?) to the old IMS and CODASYL approaches, and it applies to certain object-based approaches as well——see, e.g., the proposals of ODMG [25.11]. And XML.

———————

* And possibly as a property by still others.

———————

Caveat: Conflicts in terminology between the semantic and underlying formal levels can lead to confusion and error. In particular, "entity types" at the semantic level almost certainly do *not* map to "types"* at the formal level (forward pointer to Chapter 26, Section 26.2, if you like), and "entity supertypes and subtypes" at the semantic level almost certainly do *not* map to "supertypes and subtypes" at the formal level (forward pointer to Chapter 20, Section 20.1, if you like).

———————

* Certainly not to *scalar* types, at any rate. They might map to relation types. (Even if they do, however, it's almost certainly still the case that entity supertypes and subtypes don't map to supertypes and subtypes at the formal level——not even to *relation* supertypes and subtypes.)

———————

14.3 The E/R Model

More or less self-explanatory. Emphasize the fact that the E/R model is not the only "extended" model. Note that there are often good reasons to treat one-to-one and one-to-many relationships as if they were in fact many-to-many.

You could augment or even replace the treatment of the E/R stuff, both in this section and in the next two, by a treatment of some other approach (e.g., UML, perhaps).

14.4 E/R Diagrams

A picture is worth a thousand words ... but if so, why do we say so in words? And in any case, *which* picture? Words are, or at least can be, precise. Pictures——in a context like the one at hand——need very careful explanation (in *words!*) if errors and misconceptions are to be avoided. I have no problem with using pictures to *help* in the design process, but don't give the impression that they do the whole job.

14.5 Database Design with the E/R Model

Mostly self-explanatory. But note very carefully the suggestion that entity supertypes and subtypes are best handled by means of *views*. (There's a tacit assumption here that view updating is properly supported. If it isn't, suitable support will have to be provided "by hand" via stored or triggered procedures.)

14.6 A Brief Analysis

A somewhat contentious section ... It can be skipped if it's not to the instructor's taste. The subsections are:

- The E/R Model as a Foundation for the Relational Model?
- Is the E/R Model a Data Model?
- Entities *vs*. Relationships
- A Final Observation

The last of these asks a good rhetorical question: How would you represent an arbitrary join dependency in an E/R diagram?

References and Bibliography

References [14.22-14.24] and [14.33] are recommended.

Answers to Exercises

14.1 Semantic modeling is the activity of attempting to represent meaning.

14.2 The four steps in defining an "extended" model are as follows:

- Identify useful semantic concepts.
- Devise formal objects.
- Devise formal integrity rules ("metaconstraints").
- Devise formal operators.

14.3 See Section 14.3.

14.4

```
CONSTRAINT TUTD_14_4
    FORALL PX ( EXISTS SPX ( SPX.P# = PX.P# ) ) ;

CREATE ASSERTION SQL_14_4 CHECK (
    NOT EXISTS ( SELECT PX.* FROM P PX
                 WHERE NOT EXISTS ( SELECT SPX.* FROM SP SPX
                                    WHERE SPX.P# = PX.P# ) )
                               ) ;
```

14.5 (a) Let employees have dependents and dependents have friends, and consider the relationship between dependents and friends. (b) Let shipments be a relationship between suppliers and parts, and consider the relationship between shipments and projects. (c) Consider "large shipments," where a large shipment is one with quantity greater than 1000, say. (d) Let large shipments (only) be containerized and hence have containers as corresponding weak entities.

14.6 *No answer provided.*

14.7 *No answer provided.*

14.8 *No answer provided.*

14.9 *No answer provided.*

14.10 *No answer provided.*

***** End of Chapter 14 *****

P A R T I V

T R A N S A C T I O N M A N A G E M E N T

This part of the book contains two chapters, both of which are crucial (they mustn't be skipped). Chapter 15 discusses recovery and Chapter 16 discusses concurrency. Both describe conventional techniques in the main body of the chapter and alternative or more forward-looking ideas (e.g., multi-version controls, in the case of concurrency) in the exercises and/or the "References and Bibliography" section, and/or in the answers in this manual. *Note:* As far as possible, Chapter 15 avoids concurrency issues.

***** End of Introduction to Part IV *****

Copyright (c) 2003 C. J. Date

Chapter 15

R e c o v e r y

Principal Sections

- Transactions
- Transaction recovery
- System recovery
- Media recovery
- Two-phase commit
- Savepoints (a digression)
- SQL facilities

General Remarks

Transaction theory (which, it should immediately be said, represents a huge and very successful contribution to the database field) is, in principle, somewhat independent of the relational model. On the other hand, much transaction theory is in fact explicitly formulated in relational terms, because the relational model provides a framework that:

a. Is crystal clear and easy to understand, and

b. Allows problems to be precisely articulated and hence systematically attacked.

These remarks apply to Chapter 16 as well as the present chapter.

Recovery involves some kind of (controlled) **redundancy**. The redundancy in question is, of course, between the database *per se* and the **log**.*

---

* A nice piece of conventional wisdom: The database isn't the database; the *log* is the database, and the database is just an optimized access path to the most recent part of the log. Note the relevance of these observations to the subject of Chapter 23.

15.2 Transactions

Copyright (c) 2003 C. J. Date

Essentially standard stuff:* How to make something that's not "atomic" at the implementation level behave as if it were atomic at the model level——**BEGIN TRANSACTION, COMMIT, ROLLBACK**. Briefly discuss the **recovery log**.

———————

* Though we're going offer some rather heretical opinions on this subject in the next chapter, *q.v.*

———————

A transaction is a *unit of work*. No nested transactions until the next chapter. Important: Remind students of the difference between *consistent* and *correct* (note the relevance of the predicate stuff yet again!). Explain the place of *multiple assignment* in the overall scheme of things: If supported, transactions as a unit of work wouldn't be necessary (in theory, though they'd presumably still be useful in practice). So we'll ignore multiple assignment until further notice (= Section 16.10).

15.3 Transaction Recovery

Commit points or "synchpoints." Program execution as a *sequence* of transactions (no nesting). Implicit ROLLBACK. The **write-ahead log rule**. **ACID** properties. Explain *stealing* and *forcing;* revisit the write-ahead log rule. Group commit.

A transaction is a unit of *recovery,* a unit of *concurrency* (see the next chapter), and a unit of *integrity* (but see the next chapter).

15.4 System Recovery

Soft *vs.* hard crashes. Transaction **undo** (*backward* recovery) and **redo** (*forward* recovery)*;* **checkpoints;** system **restart; ARIES**. Forward pointer to Chapter 16 regarding not letting concurrency undermine recoverability ("we'll revisit the topic of recovery briefly in the next chapter, since——as you might expect——concurrency has some implications for recovery").

The section includes the following inline exercise: "Note that transactions that completed unsuccessfully (i.e., with a rollback) before the time of the crash don't enter into the

restart process at all (why not?)." *Answer:* Because their updates have already been undone, of course.

15.5 Media Recovery

Included for completeness. Unload/reload.

15.6 Two-Phase Commit

Don't go into too much detail, just explain the basic idea. Forward pointer to Chapter 21 on distributed databases ... but it's important to understand that "2øC"——note the fancy abbreviation!——is applicable to centralized systems, too.

15.7 Savepoints (a digression)

Self-explanatory. Concrete examples in the next section (?). *Note:* Just as an aside——this point isn't mentioned in the book——I think it was always a mistake to distinguish the operations of establishing a savepoint and committing the transaction. One consequence of making that distinction is that existing transaction source code can't be directly incorporated into some new program that has its own transaction structure. A similar remark applies to dynamic invocation, too.

15.8 SQL Facilities

Explain **START TRANSACTION** (*access mode* and *isolation level*——the latter to be discussed in detail in the next chapter). *Note:* START TRANSACTION was added to SQL in 1999. Probably ignore SET TRANSACTION (it's deliberately not mentioned in the text). Ditto for implicit START TRANSACTION.

By the way: It's odd that the SQL standards committee decided to call the operation *START* TRANSACTION, not BEGIN TRANSACTION, when they added the functionality to the standard in 1999, given that BEGIN was already a reserved word but START wasn't.

"The possibly complex repositioning code" that might be needed on the next OPEN if the cursor WITH HOLD option isn't supported is probably worth illustrating.* Use an ORDER BY based on (say) three columns (e.g., ORDER BY S#, P#, J#); the WHERE clause gets pretty horrible pretty quickly!——perhaps especially if some of the "sort columns" specify ASC and some DESC. *Note:* SQL:1999 supports the WITH HOLD option but SQL:1992 didn't.

Copyright (c) 2003 C. J. Date

---

* A simple example is given in the answer to Exercise 15.6.

Illustrate savepoints. Why didn't SQL call the operators CREATE and DROP SAVEPOINT? This is a rhetorical question, of course; I suppose the answer is that (as Hugh Darwen once remarked) it would be inconsistent to fix the inconsistencies of SQL.

References and Bibliography

Reference [15.1] is recommended as a tutorial introduction to TP monitors. References [15.4], [15.7-15.8], and [15.10] are classics, and reference [15.20] is becoming one (reference [15.10] is subsumed by the "instant classic" reference [15.12], of course). References [15.3], [15.9], and [15.16-15.17] are concerned with various "extended" transaction models; perhaps say a word on why the classical model might be unsatisfactory in certain newer kinds of application areas, especially ones involving a lot of human interaction.

Answers to Exercises

15.1 Such a feature would conflict with the objective of transaction atomicity. If a transaction could commit some but not all of its updates, then the uncommitted ones might subsequently be rolled back, whereas the committed ones of course couldn't be. Thus, the transaction would no longer be "all or nothing."

15.2 See Section 16.10, subsection "Durability."

15.3 Basically, the write-ahead log rule states that the log records for a given transaction *T* must be physically written before commit processing for *T* can complete. The rule is necessary to ensure that the restart procedure can recover any transaction that completed successfully but didn't manage to get its updates physically written to the database prior to a system crash. See Section 15.3 for further discussion.

15.4 (a) Redo is never necessary following system failure. (b) Physical undo is never necessary, and hence undo log records are also unnecessary.

15.5 For a statement of the two-phase commit protocol, see Section 15.6. For a discussion of failures during the process, see Chapter 21.

15.6 This exercise is typical of a wide class of applications, and the following outline solutions are typical too. First we show a solution without using the CHAIN and WITH HOLD features that were introduced with SQL:1999:

```
EXEC SQL DECLARE CP CURSOR FOR
         SELECT P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY
         FROM   P
         WHERE  P.P# > previous_P#
         ORDER  BY P# ;

previous_P# := '' ;
eof := FALSE ;
DO WHILE ( NOT eof ) ;
   EXEC SQL START TRANSACTION ;
   EXEC SQL OPEN CP ;
   DO count := 1 TO 10 ;
      EXEC SQL FETCH CP INTO :P#, ... ;
      IF SQLSTATE = '02000' THEN
         DO ;
            EXEC SQL CLOSE CP ;
            EXEC SQL COMMIT ;
            eof := TRUE ;
         END DO ;
      ELSE print P#, ... ;
      END IF ;
   END DO ;
   EXEC SQL DELETE FROM P WHERE P.P# = :P# ;
   EXEC SQL CLOSE CP ;
   EXEC SQL COMMIT ;
   previous_P# := P# ;
END DO ;
```

Observe that we lose position within the parts table P at the end of each transaction (even if we didn't close cursor CP explicitly, the COMMIT would close it automatically anyway). The foregoing code will therefore not be particularly efficient, because each new transaction requires a search on the parts table in order to reestablish position. Matters might be improved somewhat if there happens to be an index on the P# column——as in fact there probably will be, since {P#} is the primary key——and the optimizer chooses that index as the access path for the table.

Here by way of contrast is a solution using the new CHAIN and WITH HOLD features:

```
EXEC SQL DECLARE CP CURSOR WITH HOLD FOR
```

```
            SELECT P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY
            FROM   P
            ORDER  BY P# ;

    eof := FALSE ;
    EXEC SQL START TRANSACTION ;
    EXEC SQL OPEN CP ;
    DO WHILE ( NOT eof ) ;
       DO count := 1 TO 10 ;
          EXEC SQL FETCH CP INTO :P#, ... ;
          IF SQLSTATE = '02000' THEN
             DO ;
                EXEC SQL CLOSE CP ;
                EXEC SQL COMMIT ;
                eof := TRUE ;
             END DO ;
          ELSE print P#, ... ;
          END IF ;
       END DO ;
       EXEC SQL DELETE FROM P WHERE P.P# = :P# ;
       EXEC SQL COMMIT AND CHAIN ;
    END DO ;
```

A blow-by-blow comparison of the two solutions is left as a subsidiary exercise.

***** End of Chapter 15 *****

Copyright (c) 2003 C. J. Date

Chapter 16

C o n c u r r e n c y

Principal Sections

- Three concurrency problems
- Locking
- The three concurrency problems revisited
- Deadlock
- Serializability
- Recovery revisited
- Isolation levels
- Intent locking
- Dropping ACID
- SQL facilities

General Remarks

Very intuitive introduction: Two independently acting agents* can get in each other's way (i.e., interfere with each other)——think of, e.g., two people both trying to use the bathroom at the same time in the morning. The solution to the problem is to introduce a ***mechanism*** (door locks) and a ***protocol*** for using that mechanism (lock the bathroom door if you don't want to be disturbed).

———————

* I'm not using the term "agent" here in any special technical sense——in particular, not in the formal sense of Chapter 21.

———————

By analogy with intuitive examples such as the foregoing, concurrency control in transaction processing systems has traditionally been based on a mechanism called **locking** (though of course the locks involved are software constructs, not hardware) and a **protocol** ("the two-phase locking protocol") for using that mechanism. Moreover, most systems still typically rely on locking right up to this day, a fact that explains the emphasis on locking in the body of the chapter. However, certain nonlocking schemes are described in the annotation to several of the references in the "References and Bibliography" section.

Copyright (c) 2003 C. J. Date

16.2 Three Concurrency Problems

The three classic problems: **lost updates, uncommitted dependencies,** and **inconsistent analysis**. The examples are straightforward. Observe that the lost update problem can occur in two different ways. *Note:* Uncommitted dependencies are also called *dirty reads,* and inconsistent analysis is also called *nonrepeatable read* (though this latter term is sometimes taken to include the *phantom* problem also). Mention *conflict* terminology: RR =* no problem; RW =* inconsistent analysis / nonrepeatable read; WR =* uncommitted dependency; WW =* lost update.

16.3 Locking

Discuss only **exclusive** (X) and **shared** (S) locks at this stage. Carefully distinguish between the *mechanism* and the *protocol* (beginners often get confused over which is which; both are needed!). Explain that the whole business is usually implicit in practice.

16.4 The Three Concurrency Problems Revisited

Self-explanatory.

16.5 Deadlock

Mostly self-explanatory. Explain the **Wait-For Graph** (it isn't discussed in detail in the text because it's fairly obvious, not to say trivial; see the answer to Exercise 16.4). Detection *vs.* avoidance *vs.* timeout (perhaps skip avoidance).

16.6 Serializability

A given interleaved execution (= schedule) is considered to be correct if and only if it is equivalent to **some** serial execution (= schedule); thus, there might be several different but equally correct overall outcomes.

Discuss the **two-phase locking theorem** (important!) and the **two-phase locking protocol**.

If *A* and *B* are any two transactions in some serializable schedule, then **either B can see A's output or A can see B's**.

If transaction *A* is not two-phase, it's **always** possible to construct some other transaction *B* that can run interleaved with *A* in such a way as to produce an overall schedule that's not serializable and not correct. Real systems typically do allow transactions that aren't two-phase (see the next section but one), but allowing such a transaction——*T*, say——amounts to a gamble that no interfering transaction will ever coexist with *T* in the system. Such gambles aren't recommended! (Personally, I really question whether isolation levels lower than the maximum would ever have been supported if we'd started out with a good understanding of the importance of the integrity issue in the first place. See Section 16.8.)

16.7 Recovery Revisited

This section could be skipped. If not, explain the concept of an unrecoverable schedule, plus the sufficient conditions for recoverable and cascade-free schedules.

16.8 Isolation Levels

(Begin quote)

Serializability guarantees isolation in the ACID sense. One direct and very desirable consequence is that if all schedules are serializable, then the application programmer writing the code for a given transaction *A* need pay absolutely no attention at all to the fact that some other transaction *B* might be executing in the system at the same time. However, it can be argued that the protocols used to guarantee serializability reduce the degree of concurrency or overall system throughput to unacceptable levels. In practice, therefore, systems usually support a variety of *levels* of "isolation" (in quotes because any level lower than the maximum means the transaction isn't truly isolated from others after all, as we'll soon see).

(End quote)

As this extract indicates, I think the concept of "isolation levels" is and always was a logical mistake. But it has to be covered ... The only safe level is the highest (no interference at all), called **repeatable read** in DB2 and SERIALIZABLE——a misnomer——in SQL:1999. **Cursor stability** (this is the DB2 term——the SQL:1999 equivalent is READ COMMITTED) should also be discussed, however.* Perhaps mention U locks (partly to illustrate the point that X and S locks, though the commonest perhaps, aren't the only kind).

Copyright (c) 2003 C. J. Date

---

* I remark in passing that DB2 now supports the same four isolation levels as the SQL standard does, albeit under different names: RR or *repeatable read* ("SERIALIZABLE"), RS or *read stability* ("REPEATABLE READ"), CS or *cursor stability* ("READ COMMITTED"), and UR or *uncommitted read* ("READ UNCOMMITTED"). The terms in parentheses are the standard ones. Incidentally, DB2 allows these various options to be specified at the level of *specific database accesses* (i.e., individual SELECT, UPDATE, etc., statements).

Stress the point that *if transaction* T *operates at less than the maximum isolation level, then we can no longer guarantee that* T *if running concurrently with other transactions will transform a "correct" (consistent) state of the database into another such state.* A system that supports any isolation level lower than the maximum should provide some explicit concurrency control facilities (e.g., an explicit LOCK statement) to allow users to guarantee safety for themselves in the absence of such a guarantee from the system itself. DB2 does provide such facilities but the standard doesn't. (In fact, the standard doesn't mention locks, as such, at all——deliberately. The idea is to allow an implementation to use some nonlocking scheme if it wants to.)

Explain *phantoms* and the basic idea (only) of *predicate locking*. Mention access path locking as an implementation of predicate locking.

16.9 Intent Locking

This is good stuff (unlike the isolation level stuff!). Discuss **locking granularity** and corresponding tradeoffs. Conflict detection requires *intent locks:* **intent shared** (IS), **intent exclusive** (IX), and **shared intent exclusive** (SIX). Discuss the **intent locking protocol** (simplified version only; the full version is explained in the annotation to reference [16.10]). Mention **lock precedence** and **lock escalation**.

16.10 Dropping ACID

This section offers some fresh and slightly skeptical (unorthodox, contentious) observations on the question of the so-called *ACID properties* of transactions. You might want to skip it.

Review the intended meaning of "the ACID properties" (C for correctness, not consistency, though). We now propose to deconstruct these concepts; in fact, I believe we've all been sold a bill of goods, slightly, in this area, ***especially*** with respect to "consistency" or "correctness" ...

Begin by taking care of some unfinished business: Explain why we believe ***all constraint checking has to be immediate*** (for detailed arguments, see the book). Critical role of multiple assignment.

Now discuss the ACID properties ***per se*** (in the order C-I-D-A). Follow the arguments in the book.

- ***With respect to "C":*** Could it have been that transaction theory was worked out before we had a clear notion of consistency? (Yes, I think so.) Note the quotes from the Gray & Reuter book! Note too this text from the discussion in the chapter: "[If] the C in ACID stands for consistency, then in a sense the property is trivial; if it stands for correctness, then it's unenforceable. Either way, therefore, the property is essentially meaningless, at least from a formal standpoint."

- ***With regard to "I":*** The original term was "degrees of consistency" ... Not the happiest of names! Data is either consistent or it isn't. (Quote from the annotation to reference [16.11].)

- ***With regard to "D":*** Makes sense only if there's no nesting ... but nesting is desirable "for at least three reasons: intra-transaction parallelism, intra-transaction recovery control, and system modularity" [15.15].

- ***With regard to "A":*** Multiple assignment again!

In sum: A makes sense only because we don't have multiple assignment (but we need multiple assignment, and we already have it partially——even in SQL!——and we're going to get more of it in SQL:2003); C is only a desideratum, it can't be guaranteed; the same is true for I; and D makes sense only without nesting, but we want nesting. To quote the conclusion of this section in the book, then:

(Begin quote)

We conclude that, overall, the transaction concept is important more from a pragmatic point of view than it is from a theoretical one. Please understand that this conclusion mustn't be taken as

disparaging! We have nothing but respect for the many elegant and useful results obtained from over 25 years of transaction management research. We're merely observing that we now have a better understanding of some of the assumptions on which that research has been based——a better appreciation of integrity constraints in particular, plus a recognition of the need to support multiple assignment as a primitive operator. Indeed, it would be surprising if a change in assumptions didn't lead to a change in conclusions.

(End quote)

16.11 SQL Facilities

No explicit locking, but SQL does support isolation levels (discuss options on START TRANSACTION; recall that REPEATABLE READ in the SQL standard is not the same thing as "repeatable read" in DB2). Explain SQL's definitions of ***dirty read, nonrepeatable read,*** and ***phantoms*** (are they the same as the definitions given in the body of the chapter?). Is the SQL support broken?——see references [16.2] and [16.14].

References and Bibliography

References [16.1], [16.3], [16.7-16.8], [16.13], [16.15-16.17], and [16.20] discuss approaches to concurrency control that are wholly or partly based on something other than locking.

Answers to Exercises

16.1 See Section 16.6.

16.2 For a precise statement of the two-phase locking protocol and the two-phase locking theorem, see Section 16.6. For an explanation of how two-phase locking deals with RW, WR, and WW conflicts, see Sections 16.2-16.4.

16.3

a. There are six possible correct results, corresponding to the six possible serial schedules:

   Initially : *A* = 0
   T1-T2-T3 : *A* = 1
   T1-T3-T2 : *A* = 2
   T2-T1-T3 : *A* = 1
   T2-T3-T1 : *A* = 2

T3-T1-T2 : *A* = 4
T3-T2-T1 : *A* = 3

Of course, the six possible correct results aren't all distinct. As a matter of fact, it so happens in this particular example that the possible correct results are all independent of the initial state of the database, owing to the nature of transaction *T3*.

b. There are 90 possible distinct schedules. We can represent the possibilities as follows. (*Ri, Rj, Rk* stand for the three RETRIEVE operations *R1, R2, R3,* not necessarily in that order; similarly, *Up, Uq, Ur* stand for the three UPDATE operations *U1, U2, U3,* again not necessarily in that order.)

Ri-Rj-Rk-Up-Uq-Ur : 3 * 2 * 1 * 3 * 2 * 1 = 36 possibilities
Ri-Rj-Up-Rk-Uq-Ur : 3 * 2 * 2 * 1 * 2 * 1 = 24 possibilities
Ri-Rj-Up-Uq-Rk-Ur : 3 * 2 * 2 * 1 * 1 * 1 = 12 possibilities
Ri-Up-Rj-Rk-Uq-Ur : 3 * 1 * 2 * 1 * 2 * 1 = 12 possibilities
Ri-Up-Rj-Uq-Rk-Ur : 3 * 1 * 2 * 1 * 1 * 1 = 6 possibilities

TOTAL = 90 combinations

c. Yes. For example, the schedule *R1-R2-R3-U3-U2-U1* produces the same result (one) as two of the six possible serial schedules (*Exercise:* Check this statement), and thus happens to be "correct" for the given initial value of zero. But it must be clearly understood that this "correctness" is a mere fluke, and results purely from the fact that the initial data value happened to be zero and not something else. As a counterexample, consider what would happen if the initial value of *A* were ten instead of zero. Would the schedule *R1-R2-R3-U3-U2-U1* shown above still produce one of the genuinely correct results? (What *are* the genuinely correct results in this case?) If not, then that schedule isn't serializable.

d. Yes. For example, the schedule *R1-R3-U1-U3-R2-U2* is serializable (it's equivalent to the serial schedule *T1-T3-T2*), but it cannot be produced if *T1, T2,* and *T3* all obey the two-phase locking protocol. For, under that protocol, operation *R3* will acquire an S lock on *A* on behalf of transaction *T3;* operation *U1* in transaction *T1* will thus not be able to proceed until that lock has been released, and that won't happen until transaction *T3* terminates (in fact, transactions *T3* and *T1* will deadlock when operation *U3* is reached).

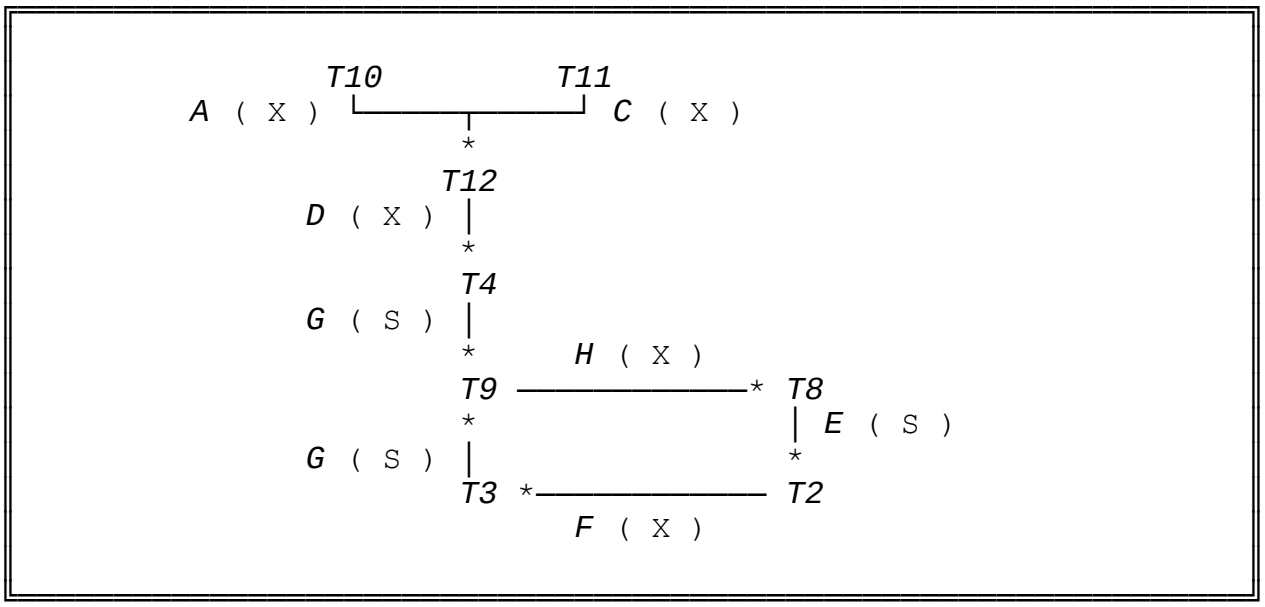
This exercise illustrates very clearly the following important point. Given a set of transactions and an initial state of the

Copyright (c) 2003 C. J. Date

database, (a) let ALL be the set of all possible schedules involving those transactions; (b) let "CORRECT" be the set of all schedules that do at least produce a correct final state from the given initial state; (c) let SERIALIZABLE be the set of all guaranteed correct (i.e., serializable) schedules; and (d) let PRODUCIBLE be the set of all schedules producible under the two-phase locking protocol. Then, in general,

ALL ⊇ "CORRECT" ⊇ SERIALIZABLE ⊇ PRODUCIBLE

16.4 At time *tn* no transactions are doing any useful work at all! There's one deadlock, involving transactions *T2, T3, T9,* and *T8;* in addition, *T4* is waiting for *T9, T12* is waiting for *T4,* and *T10* and *T11* are both waiting for *T12.* We can represent the situation by means of a graph (the **Wait-For Graph**), in which the nodes represent transactions and a directed edge from node *Ti* to node *Tj* indicates that *Ti* is waiting for *Tj* (see the figure below). Edges are labeled with the name of the database item and level of lock they're waiting for.



16.5 Isolation level CS has the same effect as isolation level RR on the problems of Figs. 16.1-16.3. (Note, however, that this statement does *not* apply to CS as implemented in DB2, thanks to DB2's use of U locks in place of S locks [4.21].) As for the inconsistent analysis problem (Fig. 16.4): Isolation level CS doesn't solve this problem; transaction *A* must execute under RR in order to retain its locks until end-of-transaction, for otherwise it'll still produce the wrong answer. (Alternatively, of course, *A* could lock the entire accounts relvar via some explicit lock request, if the system supports such an operation. This solution would work under both CS and RR isolation levels.)

Copyright (c) 2003 C. J. Date

16.6 See Section 16.9. Note in particular that the *formal* definitions are given by the lock type compatibility matrix (Fig. 16.13).

16.7 See Section 16.9.

16.8 See the annotation to reference [16.10].

16.9 The three concurrency problems identified in Section 16.2 were *lost update, uncommitted dependency,* and *inconsistent analysis*. Of these three:

- *Lost updates:* The SQL implementation is required to guarantee (in all circumstances) that lost updates never occur.

- *Uncommitted dependency:* This is just another name for dirty read.

- *Inconsistent analysis:* This term covers both nonrepeatable read and phantoms.

16.10 The following brief description is based on one given in reference [15.6]. First of all, the system must keep:

1. For each data object, a stack of committed versions (each stack entry giving a value for the object and the ID of the transaction that established that value; i.e., each stack entry essentially consists of a pointer to the relevant entry in the log). The stack is in reverse chronological sequence, with the most recent entry being on the top.

2. A list of transaction IDs for all committed transactions (the *commit list*).

When a transaction starts executing, the system gives it a private copy of the commit list. Read operations on an object are directed to the most recent version of the object produced by a transaction on that private list. Write operations, by contrast, are directed to the actual current data object (which is why write/write conflict testing is still necessary). When the transaction commits, the system updates the commit list and the data object version stacks appropriately.

***** End of Chapter 16 *****

Copyright (c) 2003 C. J. Date

P A R T V

F U R T H E R T O P I C S

The introduction to Part V in the book itself says it all:

(Begin quote)

We claimed in Part II of this book that the relational model is the foundation for modern database technology, and so it is. However, it's *only* the foundation: There's a lot more to database technology than just the relational model as described in Part II, and database students and professionals need to be familiar with many additional concepts and facilities in order to be fully "database-aware" (as indeed should be obvious from our discussions in Parts III and IV). We now turn our attention to a miscellaneous collection of further important topics. The topics to be covered, in sequence, are as follows:

- Security (Chapter 17)
- Optimization (Chapter 18)
- Missing information (Chapter 19)
- Type inheritance (Chapter 20)
- Distributed databases (Chapter 21)
- Decision support (Chapter 22)
- Temporal data (Chapter 23)
- Logic-based databases (Chapter 24)

Actually the foregoing sequence is a little arbitrary, but the chapters have been written on the assumption that they'll be read (possibly selectively) in order as written.

(End quote)

As this quote indicates, it's up to you, somewhat, to decide which of these chapters you want to cover and which skip. Further guidance is included in the notes on each chapter.

Copyright (c) 2003 C. J. Date

***** End of Introduction to Part V *****

Chapter 17

S e c u r i t y

Principal Sections

- Discretionary access control
- Mandatory access control
- Statistical DBs
- Data encryption
- SQL facilities

General Remarks

Security can be defined as *the protection of data against unauthorized access* (i.e., unauthorized disclosure, alteration, or destruction). Regarding the superficial similarities between security and integrity: **Integrity** is paramount (it's part of the foundation); **security,** by contrast, is secondary (it *builds on* the foundation, but isn't itself part of that foundation).

Note: When I say security is secondary, of course I don't mean to suggest it's not important. On the contrary, it's very important, and becoming more so, especially in these days of the Internet and e-commerce. But it's secondary from a database foundations point of view.

Explain **discretionary** *vs*. **mandatory** control. Mention **authentication** and **user groups** (also called *roles*——see Section 17.6).

Any or all of Sections 17.3-17.6 can be skipped if desired. Section 17.2 probably shouldn't be skipped, however.

17.2 Discretionary Access Control

This kind of access control is the one most commonly found in practice. It's supported by SQL (through the GRANT and REVOKE statements——see Section 17.6). The term **authority** doesn't exactly have an SQL equivalent, though; what's more, SQL's "privileges" (its counterpart to authorities) have no name! (Makes them difficult to talk about, and in fact led to a bug in the original System R implementation [17.10].)

Copyright (c) 2003 C. J. Date

The section should be more or less self-explanatory. Explain the Ingres request modification scheme (relate to view processing, and possibly to integrity enforcement too). Mention audit trails.

17.3 Mandatory Access Control

There's quite a lot in the research literature on **"multi-level security"** (MLS). Explain the basic idea.

The section includes the following inline exercise: "Do you think the [relational MLS] ideas ... constitute a violation of *The Information Principle?* Justify your answer!" *Answer:* No, they don't. Polyinstantiated relvars are analogous, somewhat, to *views,* though with view-specific defaults. Note that there's presumably at least one user——the DBA, or perhaps the *security administrator?*——who's allowed to see the hidden classification levels.

17.4 Statistical DBs

Most database textbooks have comparatively little to say on this topic, but the growing interest in data warehouses (see Chapter 22) makes the material increasingly relevant. The details of individual and general **trackers,** etc., are interesting and straightforward, if somewhat depressing. (By the way, there was a bug in this area in the 7th edition, but it's been corrected now. Thanks to Guy de Tré of Ghent University for drawing my attention to the bug in question.)

17.5 Data Encryption

Self-explanatory. The point's worth making that the biggest threat to security isn't the database or the DBMS but the communications network (wireless or otherwise). Mention the DES and the AES. Emphasize the **public key** stuff (the **RSA algorithm** in particular). Explain **digital signatures**.

The section includes an example in which the plaintext string

AS KINGFISHERS CATCH FIRE

is encrypted as follows:

FDIZBSSOXLMQ GTHMBRAERRFY

It then asks the reader to decrypt this ciphertext. *Answer:* Obvious. But the real question is, can you figure out the

encryption key, given just the ciphertext? (Not too easy.) Or given both the ciphertext and the plaintext? (Trivial.)
Subsidiary question: Do you recognize the source of the quote?

17.6 SQL Facilities

Self-explanatory again, except perhaps for the *grant option* stuff and RESTRICT *vs.* CASCADE (the "abandoned privileges" stuff). Briefly explain the less commonly encountered privileges USAGE, UNDER, REFERENCES, TRIGGER, and EXECUTE. Note that GRANT and REVOKE are really CREATE and DROP AUTHORITY, but because the syntax is different those "authorities" are unnamed.

The text includes the following example of a "context-dependent" authority in SQL:

```
CREATE VIEW S_NINE_TO_FIVE AS
       SELECT S.S#, S.SNAME, S.STATUS, S.CITY
       FROM   S
       WHERE  CURRENT_TIME ≥ TIME '09:00:00'
       AND    CURRENT_TIME ≤ TIME '17:00:00' ;

GRANT SELECT, UPDATE ( STATUS )
ON    S_NINE_TO_FIVE
TO    ACCOUNTING ;
```

It then goes to point out that S_NINE_TO_FIVE is "rather an odd kind of view!——its value changes over time, even if the underlying data does not." And it asks what the predicate is for this view.
Answer: Without going into details, it should be clear that the predicate in question must include "the current time" as one of its parameters (and instantiating that predicate must involve substituting the actual "current time" for that parameter). By contrast, the predicate for what we might call a "regular" relvar (regardless of whether it's base or derived) includes only parameters that stand for *values recorded in the database*.

Answers to Exercises

17.1

```
a. AUTHORITY AAA
      GRANT RETRIEVE ON STATS TO Ford ;

b. AUTHORITY BBB
      GRANT INSERT, DELETE ON STATS TO Smith ;

c. AUTHORITY CCC
```

```
      GRANT RETRIEVE
      ON    STATS
      WHEN  USER () = NAME
      TO    ALL ;
```

We're assuming here that users use their own name as their user ID. Note the use of a WHEN clause and the niladic built-in operator USER().

d.
```
AUTHORITY DDD
   GRANT RETRIEVE, UPDATE { SALARY, TAX }
   ON    STATS
   TO    Nash ;
```

e.
```
AUTHORITY EEE
   GRANT RETRIEVE { NAME, SALARY, TAX }
   ON    STATS
   TO    Todd ;
```

f.
```
AUTHORITY FFF
   GRANT RETRIEVE { NAME, SALARY, TAX },
         UPDATE { SALARY, TAX }
   ON    STATS
   TO    Ward ;
```

g.
```
VAR PREACHERS VIEW
    STATS WHERE OCCUPATION = 'Preacher' ;

AUTHORITY GGG
    GRANT ALL
    ON    PREACHERS
    TO    Pope ;
```

Note the need to use a view in this example.

h.
```
VAR NONSPECIALIST VIEW
    WITH ( STATS RENAME OCCUPATION AS X ) AS T1,
         ( EXTEND STATS
           ADD COUNT ( T1 WHERE X = OCCUPATION ) AS Y ) AS T2,
         ( T2 WHERE Y > 10 ) AS T3 :
           T3 { ALL BUT Y }

AUTHORITY HHH
    GRANT DELETE
    ON    NONSPECIALIST
    TO    Jones ;
```

i.
```
VAR JOBMAXMIN VIEW
WITH ( STATS RENAME OCCUPATION AS X ) AS T1,
     ( EXTEND STATS ADD
       ( MAX ( T1 WHERE X = OCCUPATION, SALARY ) AS MAXSAL,
```

```
            MIN ( T1 WHERE X = OCCUPATION, SALARY ) AS MINSAL ) )
                                              AS T2 :
    T2 { OCCUPATION, MAXSAL, MINSAL }

    AUTHORITY III
        GRANT RETRIEVE ON JOBMAXMIN TO King ;
```

17.2 We make just one observation here: A user who has the authority to define a new base relvar and in fact does so can be regarded as the **owner** of that new relvar. As in SQL, the owner of a given base relvar should automatically be granted all possible privileges on that relvar,* including not only the RETRIEVE, INSERT, UPDATE, and DELETE privileges (of course), but also the privilege of defining authorities that grant privileges on that relvar to other users.

---

* Some might disagree with this statement, arguing that it should be possible for one user (perhaps the DBA) to be able to define a new base relvar on some other user's behalf and *not* be automatically granted all privileges on the relvar in question.

17.3 An individual tracker for Hal is

```
    CHILDREN > 1 AND NOT ( OCCUPATION = 'Homemaker' )
```

Consider the following sequence of queries:

```
    COUNT ( STATS WHERE CHILDREN > 1 )
```

Result: 6.

```
    COUNT ( STATS WHERE CHILDREN > 1 AND NOT
                      ( OCCUPATION = 'Homemaker' ) )
```

Result: 5.

Hence the expression

```
    CHILDREN > 1 AND OCCUPATION = 'Homemaker'
```

uniquely identifies Hal.

```
    SUM ( STATS WHERE CHILDREN > 1, TAX )
```

Result: 48K.

```
SUM ( STATS WHERE CHILDREN > 1 AND NOT
                ( OCCUPATION = 'Homemaker' ), TAX )
```

Result: 46K.

Hence Hal's tax figure is 2K.

17.4 General tracker: SEX = 'F'.

```
SUM ( STATS WHERE SEX = 'F', TAX )
```

Result: 70K.

```
SUM ( STATS WHERE NOT ( SEX = 'F' ), TAX )
```

Result: 16K.

Hence the total tax is 86K.

```
SUM ( STATS WHERE ( CHILDREN > 1 AND
                    OCCUPATION = 'Homemaker' ) OR
                  SEX = 'F', TAX )
```

Result: 72K.

```
SUM ( STATS WHERE ( CHILDREN > 1 AND
                    OCCUPATION = 'Homemaker' ) OR NOT
                  ( SEX = 'F' ), TAX )
```

Result: 16K.

Adding these results and subtracting the total previously calculated, we have Hal's tax figure = 88K - 86K = 2K.

17.5 The plaintext is

```
EYES I DARE NOT MEET IN DREAMS
```

What is the encryption key? *No answer provided. Subsidiary question:* Do you recognize the source of the quote?

17.6 *No answer provided.*

17.7 One problem is that, even in a system that supports encryption, data might still have to be processed in its plaintext form internally (e.g., for comparisons to operate correctly), and there might thus still be a risk of sensitive data being accessible to concurrently executing applications or appearing in

a memory dump. Also, there are severe technical problems in indexing encrypted data and in maintaining log records for such data (but see reference [17.4]).

17.8

a. GRANT SELECT ON STATS TO Ford ;

b. GRANT INSERT, DELETE ON STATS TO Smith ;

c.
```
CREATE VIEW MINE AS
        SELECT STATS.*
        FROM   STATS
        WHERE  STATS.NAME = CURRENT_USER ;

GRANT SELECT ON MINE TO PUBLIC ;
```

We're assuming here that users use their own name as their user ID. Note the use of the niladic built-in operator CURRENT_USER.

d. GRANT SELECT, UPDATE (SALARY, TAX) ON STATS TO Nash ;

e. GRANT SELECT (NAME, SALARY, TAX) ON STATS TO Todd ;

f.
```
GRANT SELECT ( NAME, SALARY, TAX ),
      UPDATE ( SALARY, TAX ) ON STATS TO Ward ;
```

g.
```
CREATE VIEW PREACHERS AS
        SELECT STATS.*
        FROM   STATS
        WHERE  STATS.OCCUPATION = 'Preacher' ;

GRANT ALL PRIVILEGES ON PREACHERS TO Pope ;
```

Observe the use of the shorthand "ALL PRIVILEGES" in this example. ALL PRIVILEGES in SQL doesn't literally mean all privileges, however——it means all privileges on the relevant object for which the user issuing the GRANT has grant authority.

h.
```
CREATE VIEW NONSPECIALIST AS
        SELECT STX.*
        FROM   STATS AS STX
        WHERE  ( SELECT COUNT(*)
                 FROM   STATS AS STY
                 WHERE  STY.OCCUPATION = STX.OCCUPATION ) > 10 ;

GRANT DELETE ON NONSPECIALIST TO Jones ;
```

i.
```
CREATE VIEW JOBMAXMIN AS
```

```
          SELECT STATS.OCCUPATION,
                   MAX ( STATS.SALARY ) AS MAXSAL,
                    MIN ( STATS.SALARY ) AS MINSAL
          FROM   STATS
          GROUP  BY STATS.OCCUPATION ;

   GRANT SELECT ON JOBMAXMIN TO King ;
```

17.9

a. `REVOKE SELECT ON STATS FROM Ford RESTRICT ;`

b. `REVOKE INSERT, DELETE ON STATS FROM Smith RESTRICT ;`

c. `REVOKE SELECT ON MINE FROM PUBLIC RESTRICT ;`

d.
```
REVOKE SELECT, UPDATE ( SALARY, TAX )
                         ON STATS FROM Nash RESTRICT ;
```

e.
```
REVOKE SELECT ( NAME, SALARY, TAX ) ON STATS
FROM Todd RESTRICT ;
```

f.
```
REVOKE SELECT ( NAME, SALARY, TAX ), UPDATE ( SALARY, TAX )
ON STATS FROM Ward RESTRICT ;
```

g. `REVOKE ALL PRIVILEGES ON PREACHERS FROM Pope RESTRICT ;`

h. `REVOKE DELETE ON NONSPECIALIST FROM Jones RESTRICT ;`

i. `REVOKE SELECT ON JOBMAXMIN FROM King RESTRICT ;`

***** End of Chapter 17 *****

Copyright (c) 2003 C. J. Date

Chapter 18

O p t i m i z a t i o n

Principal Sections

- A motivating example
- An overview of query processing
- Expression transformation
- DB statistics
- A divide-and-conquer strategy
- Implementing the relational operators

General Remarks

No "SQL Facilities" section in this chapter. However, some SQL-specific optimization issues are discussed in the annotation to several of the references (especially references [18.37-18.43]). Also, the summary section mentions the fact that 3VL, SQL's (flawed) support for 3VL, and duplicate rows all serve as ***optimization inhibitors*** (and the same is true for SQL's left-to-right column ordering, though this last one isn't mentioned in the chapter itself). The articles mentioned under reference [4.19] are also relevant.

The material of this chapter is stuff that——in principle, given a perfect system——the user really shouldn't need to know about. It's part of the implementation, not part of the model. However, just as a knowledge of what goes on under the hood can help you be a better driver, a knowledge of what's involved in executing queries might help you use the system better. In any case, it's interesting stuff!——and it's a major part of relational technology in general, though not part of the relational model ***per se***. The chapter really shouldn't be omitted, but it might be downplayed a little (though it goes against the grain to say so).

There are two broad aspects to optimization: **expression transformation** (*aka* "query rewrite") and **access path selection** (using indexes and other storage structures appropriately to get to the stored data). The relational model is directly relevant to the first aspect, inasmuch as it's the formal properties of the relational algebra that make expression transformation possible in the first place. It's not so directly relevant to the second aspect, except inasmuch as its clean logical ***vs***. physical separation is what permits so many different access paths to be deployed (physical data independence). Commercial optimizers do a

Copyright (c) 2003 C. J. Date

reasonably good job on the second aspect* but, strangely, not such a good job on the first——even though there's a wealth of relevant material available in the technical literature, going back as far as the early 1970s.

———————

* Though there's always room for improvement! Don't give the students the impression that today's optimizers are perfect. What's more, there's a possibility that we're on the threshold of some radical developments in this area (see Appendix A)——where by "radical" I mean that access path selection, as such, ***might no longer be needed!*** I'll say a little more about these developments in a few moments.

———————

The activities of the ***TPC*** should at least be mentioned (see the annotation to reference [18.5]). Some brief discussion of ***parallelism*** should also be included (see the annotation to reference [18.56]; forward pointer to distributed databases perhaps?). Perhaps mention too the possibility of holding the entire database in ***main memory*** (see, e.g., reference [18.50])——a possibility that changes the implementation (and optimization) picture dramatically! Also, be aware of the following remarks (they're from the end of Section 18.8, the summary section):

(Begin quote)

In this chapter, we've discussed optimization as conventionally understood and conventionally implemented; in other words, we've described "the conventional wisdom." More recently, however, a radically new approach to DBMS implementation has emerged, an approach that has the effect of invalidating many of the assumptions underlying that conventional wisdom. As a consequence, many aspects of the overall optimization process can be simplified (even eliminated entirely, in some cases), including:

- The use of ***cost-based optimizing*** (Stages 3 and 4 of the process)
- The use of ***indexes*** and other conventional access paths
- The choice between ***compiling*** and ***interpreting*** database requests
- The algorithms for ***implementing the relational operators***

and many others. See Appendix A for further discussion.

(End quote)

We live in exciting times!

18.2 A Motivating Example

Self-explanatory. Drive the message home by stressing the "human factors" aspect: If the original unoptimized query took three hours to run, the final version will run in just over ***one second***.

It can reasonably be argued that relational systems stand or fall on the basis of how good their optimizer is. (Though I do have to admit that there's at least one extremely successful commercial SQL product that managed to survive for years with what might be called "the identity optimizer." I can't explain this fact on technical grounds. I strongly suspect the explanation isn't a technical one at all.)

18.3 An Overview of Query Processing

The four stages (refer to Fig. 18.1):

1. Cast the query into internal form
2. Convert to canonical form
3. Choose candidate low-level procedures
4. Generate query plans and choose the cheapest

In practice, ***Stage 1*** effectively becomes "convert the SQL query to a relational algebra equivalent" (and that's what real commercial optimizers typically do——see, e.g., the classic paper by Selinger ***et al***. [18.33]). The obvious question arises: Why wasn't the original query stated in algebraic form in the first place? A good question ... Could perhaps sidetrack to review the origins of SQL here, and the current ironical situation, if the instructor is familiar with this story and wants to discuss it (see reference [4.16]).

View processing ("query modification") is also done during Stage 1.

Stage 2: This is the "query rewrite" stage. Elaborated in Section 18.4. Explain the general notion of **canonical form** (either here or in that later section), if you haven't already done so at some earlier point in the class.

Stage 3: The first stage at which physical storage structures, data value distributions, etc. (and hence catalog access), become relevant. Elaborated in Sections 18.5 and 18.7 (note, however, that students are expected to have a basic understanding of file structures, indexes, etc., already; you might mention the online Appendix D in this regard, and possibly even set it as a reading assignment). Stage 3 flows into Stage 4 (in fact, Fig. 18.1 doesn't distinguish between Stages 3 and 4).

Stage 4: "Access path selection" (important term!——though it's sometimes used to encompass Stage 3 as well). Elaborated in Sections 18.5 and 18.7. "Choose the cheapest plan": What's involved in evaluating the cost formulas? Need estimates of intermediate result sizes ... Estimating those sizes is a difficult problem, in general. Are optimizers only as good as their estimating? (No, they're not, but good estimating ***is*** important.) *Note:* Early experience with System R showed that the optimizer's estimates were often wildly wrong but that ***it didn't matter,*** in the sense that the plan the optimizer thought was cheapest was in fact the cheapest, the plan it thought was the second cheapest was in fact the second cheapest, and so on.* I can't begin to explain this state of affairs; perhaps it was just a fluke, and insufficient measurement of the optimizer was done. I still think good estimating is important.

———————

* I hope I'm remembering this right; I can't track down the original source.

———————

18.4 Expression Transformation

The section begins: "In this section we describe some transformation laws or rules that might be useful in Stage 2 of the optimization process. Producing examples to illustrate the rules and deciding exactly why they might be useful are both left (in part) as exercises." *No answer provided* (other than what's in the book).

Explain **distributivity, commutativity, associativity*——also **idempotence** and **absorption** (see Exercise 18.5 re the last of these concepts). Theory is practical! Also discuss transformation of other kinds of expressions——arithmetic expressions, boolean expressions, etc. *Note to the instructor:* What are the implications of user-defined types and object/relational systems on these ideas? See Chapter 26, Section 26.4. See also the

annotation to reference [25.40] for a note regarding the implications for object systems.

---

* I note in passing that some writers seem to confuse these terms, using (e.g.) commutativity to mean distributivity.

Discuss **semantic** transformations——not much supported yet in current products, but they *could* be, and the potential payoff is enormous. Declarative integrity support is crucial! (So what are object systems going to do?)

18.5 Database Statistics

Self-explanatory, pretty much (but see Exercise 18.15). Be sure to mention RUNSTATS (or some counterpart to RUNSTATS).

18.6 A Divide-and-Conquer Strategy

This section describes the historically important Ingres *query decomposition* approach and can serve as a springboard for getting into specifics of other tricks and techniques. The section might be skipped, but in any case it's fairly self-explanatory (it does use QUEL as a basis, but QUEL is easy to understand). It might be set as a reading assignment.

18.7 Implementing the Relational Operators

To quote: "[The] primary reason for including this material is simply to remove any remaining air of mystery that might possibly still surround the optimization process." In other words, the implementation techniques to be described are all pretty much what you might expect——it's all basically common sense. But see also the annotation to, e.g., references [18.9-18.15] at the end of the chapter.

The following inline exercises are included in this section:

- Give pseudocode procedures for project, summarize, and many-to-one merge join.
- Derive cost estimates for hash lookup and hash join.

These exercises can be used as a basis for class discussion. *No answers provided*.

References and Bibliography

References [18.2] and [18.3] are both recommended; either would make a good handout, though of course the latter is more recent and thus preferable. Reference [18.4] is, in my opinion, a much overlooked classic (the book in which it appears is likely to be hard to find, unfortunately; perhaps the original IBM Research Report——IBM reference number RJ1072, dated July 27th, 1972——could be tracked down?). References [18.37-18.41] illustrate some of the *implementation* difficulties caused by duplicates and nulls in SQL! (By contrast, the book as such——i.e., the eighth edition, especially Chapter 19——concentrates on the *model* or *conceptual* difficulties caused by such things.)

Answers to Exercises

18.1 a. Valid. b. Valid. c. Valid. d. Not valid. e. Valid. f. Not valid (it would be valid if we replaced the AND by an OR). g. Not valid. h. Not valid. i. Valid.

18.2 This exercise and the next overlap considerably with Exercise 7.4, *q.v.* INTERSECT is a special case of JOIN, so we can ignore it. The commutativity of UNION and JOIN is obvious from the definitions, which are symmetric in the two relations concerned. The proof that MINUS isn't commutative is trivial.

18.3 As already noted, this exercise and the previous one overlap considerably with Exercise 7.4, *q.v.* INTERSECT is a special case of JOIN, so we can ignore it. The associativity of UNION is shown in the answer to Exercise 7.4; the proof that JOIN is associative is analogous. The proof that MINUS isn't associative is trivial.

18.4 We show that (a) UNION distributes over INTERSECT. The proof that (b) INTERSECT distributes over UNION is analogous.

- If *t* ε *A* UNION (*B* INTERSECT *C*), then *t* ε *A* or *t* ε (*B* INTERSECT *C*).
  - If *t* ε *A*, then *t* ε *A* UNION *B* and *t* ε *A* UNION *C* and hence *t* ε (*A* UNION *B*) INTERSECT (*A* UNION *C*).

- If *t* ε *B* INTERSECT *C*, then *t* ε *B* and *t* ε *C*, so *t* ε *A* UNION *B* and *t* ε *A* UNION *C* and hence (again) *t* ε (*A* UNION *B*) INTERSECT (*A* UNION *C*).

- Conversely, if *t* ε (*A* UNION *B*) INTERSECT (*A* UNION *C*), then *t* ε *A* UNION *B* and *t* ε *A* UNION *C*. Hence *t* ε *A* or *t* ε both of *B* and *C*. Hence *t* ε *A* UNION (*B* INTERSECT *C*).

18.5 We show that *A* UNION (*A* INTERSECT *B*) ≡ *A*. If *t* ε *A* then clearly *t* ε *A* UNION (*A* INTERSECT *B*). Conversely, if *t* ε *A* UNION (*A* INTERSECT *B*), then *t* ε *A* or *t* ε both of *A* and *B*; either way, *t* ε *A*. The proof that *A* INTERSECT (*A* UNION *B*) ≡ *A* is analogous.

18.6 The two conditional cases were covered in Section 18.4. The unconditional cases are straightforward. We show that projection fails to distribute over difference by giving the following counterexample. Let *A*{*X*,*Y*} and *B*{*X*,*Y*} each contain just one tuple——namely, the tuples {*X x*,*Y y*} and {*X x*,*Y z*}, respectively (*y* =/ *z*). Then (*A* MINUS *B*){*X*} gives a relation containing just the tuple {*X x*}, while *A*{*X*} MINUS *B*{*X*} gives an empty relation.

18.7 We don't give a detailed answer to this exercise, but here are the kinds of questions you should be asking yourself: Can a sequence of extends be combined into a single operation? Is an extend followed by a restrict the same as a restrict followed by an extend? Does extend distribute over union? Over difference? What about summarize? And so on.

18.8 *No answer provided.*

18.9 A good set of such rules can be found in reference [18.2].

18.10 A good set of such rules can be found in reference [18.2].

18.11

a. Get "nonLondon" suppliers who do not supply part P2.

b. Get the empty set of suppliers.

c. Get "nonLondon" suppliers such that no supplier supplies fewer kinds of parts.

d. Get the empty set of suppliers.

e. No simplification possible.

f. Get the empty set of pairs of suppliers.

Copyright (c) 2003 C. J. Date

g. Get the empty set of parts.

h. Get "nonParis" suppliers such that no supplier supplies more kinds of parts.

Note that certain queries——to be specific, queries b., d., f., and g.——can be answered directly from the constraints.

18.12 *No answer provided.*

18.13 This exercise could form the basis of a simple class project; the results might even be publishable! *No answer provided.*

18.14 *No answer provided.*

18.15 For processing reasons, the true highest and/or lowest value is sometimes some kind of *dummy* value——e.g., the highest "employee name" might be a string of all Z's, the lowest might be a string of all blanks. Estimates of (e.g.) the average increment from one column value to the next in sequence would be skewed if they were based on such dummy values.

18.16 Such hints might be useful in practice, but in my opinion they amount to an abdication of responsibility on the part of the optimizer (or of the vendor, rather). Users shouldn't have to get involved in performance issues at all! Note the implications for portability, too (or lack thereof, rather). *Note:* In the particular case at hand (OPTIMIZE FOR *n* ROWS), it seems likely that what's really required is some kind of *quota query* functionality. See reference [7.5].

18.17 This exercise can be used as a basis for class discussion. *No answer provided.*

18.18 *No answer provided.*

***** End of Chapter 18 *****

Chapter 19

Missing Information

Principal Sections

- An overview of the 3VL approach
- Some consequences of the foregoing scheme
- Nulls and keys
- Outer join (a digression)
- Special values
- SQL facilities

General Remarks

Missing information is an important problem, but nulls and 3VL (= three-valued logic) are NOT a good solution to that problem; in fact, they're a disastrously bad one. However, it's necessary to discuss them,* owing to their ubiquitous nature out there in the commercial world (and, regrettably, in the research world also, at least to some extent). This chapter shouldn't be skipped, though it could perhaps be condensed somewhat.

---

* At least, I think it is. But I suppose you could just say to the students "Trust me, don't ever use nulls"; perhaps suggest a reading assignment; and move on quickly to the next topic.

Note: By "the commercial world" (and, somewhat, "the research world" as well) in the previous paragraph, what I really mean is the *SQL* world, of course.

Now, you might be aware that I've been accused of "conducting a tirade against nulls." Guilty as charged! But it's not just me; in fact, I don't know *anyone* working in the database field who both (a) fully understands nulls **and** (b) thinks they're a good idea. The fact is, not only are nulls a very *bad* idea, the full extent of their awfulness is still not widely enough appreciated in the community at large, and so the tirade seems to be necessary. (As the preface says, this is a textbook with an

Copyright (c) 2003 C. J. Date

attitude.) It's odd, really; there seems to be so much interest in "semantic modeling" out there in the community at large; and yet there seems to be comparatively little interest in the question of how to deal properly with missing information. Surely the latter is directly and highly relevant to the former? And if people really looked at the latter, they'd quickly come to appreciate some of the problems with nulls.

Anyway, the crucial point is this: **A system that supports nulls produces wrong answers**. Of course, it probably produces right answers too; but since we have no way of knowing which ones are right and which wrong, **all** answers become suspect). It's important to understand too that it's not just queries that *directly involve* nulls that can give wrong answers——*all* answers become suspect if nulls are permitted in *any* relvar in the database [19.19].

Here are some show-stopping facts that don't seem to be generally understood:

- Nulls aren't values (SQL falls down on this one, in part); thus, the frequently heard term "null value" (as opposed to just "null") is a contradiction in terms.
- A domain that "contains a null" isn't a domain.
- An attribute that "contains a null" isn't an attribute.
- A tuple that "contains a null" isn't a tuple. (It isn't a tuple at all, let alone being one that represents an instantiation of the applicable predicate.)
- A relation that "contains a null" isn't a relation.

Just why each of the foregoing assertions is valid is left as a (trivial) exercise for the reader. *Hint:* Appeal to the definitions!

The argument made in the annotation to reference [19.11], regarding the number of logical operators required for *n*VL, is worth calling out explicitly (see Exercise 19.6). As for the questions——

- What's a suitable set of *primitive* operators?
- What's a suitable set of *useful* operators?

(for 3VL in particular)——let alone the obvious follow-on questions regarding proof, correctness, implementation, usability, and so

forth——well, these are questions I've never heard the 3VL advocates even ask, let alone answer. (Note that if you want to support 3VL, then you *must* support all 19,000 plus operators; otherwise you aren't supporting 3VL. I don't know what it might be that you *are* supporting in such a case, but it isn't 3VL.)

"It all makes sense if you squint a little and don't think too hard" [11.10]. Yes, that's right. Nulls do look attractive at first, at least superficially. It's only when you start poking into them in detail that you begin to discover the problems.

19.2 An Overview of the 3VL Approach

In this section I "suspend disbelief" and explain how the 3VL approach is supposed to work.* Note the use of the terms *UNK* (i.e., null) and *unk* ("the third truth value"). They're not the same thing! (though SQL thinks they are)——another logical difference here, in fact. Explain the need for the *IS_UNK* and *MAYBE* logical operators.

———————

* Emphasis on *supposed to*. SQL doesn't fully abide by the principles laid down in this section; note carefully, however, that this fact doesn't constitute a flaw in three-valued logic *per se,* but rather a flaw in SQL's attempt to implement that logic.

———————

By the way, note the slightly tricky formulation involved in the SQL counterpart to the MAYBE example (i.e., the "AND NOT" bit); SQL does support MAYBE directly, via its IS UNKNOWN construct, but the whole point of the example is to demonstrate the "need" to support MAYBE. In other words, if you really want to support 3VL, then you need to support MAYBE. (Of course, I don't want to support 3VL at all, but I do think if you're going to do it, then you should do it right, and doing it right includes supporting MAYBE.)

Explain the effect on the quantifiers (SQL falls down here, too) and the relational operators. Note the strange——I would say completely untenable——definition of *"duplicates"* in particular! Discuss the effect on integrity constraints.

19.3 Some Consequences of the Foregoing Scheme

Recall the following from Chapter 6 of this manual (slightly reworded here), and note the final sentence in particular:

> ***Assignment doesn't always mean assignment:*** Suppose X3 is of type CHAR(3) and we assign the string 'AB' to it. After that assignment, the value of X3 is actually 'AB ' (note the blank), and the comparison X3 = 'AB' won't give ***true*** if NO PAD is in effect (see reference [4.20]). ***Note:*** Many similar but worse situations arise when nulls are taken into account (see Chapter 19).

Here's one thing that happens when nulls are taken into account:

> ***Assignment doesn't always mean assignment:*** Suppose X and Y are variables, suppose Y is null, and we assign Y to X. After that assignment, X too is null, but the comparison X = Y won't give ***true***.

To continue: Discuss the implications for expression transformation (and hence for both the optimizer and the user). Regarding ***the departments and employees example,*** observe that:

a. Optimizations that are valid in 2VL might not be valid in 3VL.

b. More important, **answers that are correct according to the logic might not be correct in the real world** ("the interpretation issue").

The database doesn't contain "the real world" but only ***knowledge about*** the real world (this obvious but important point is true regardless of whether we're dealing with nulls and 3VL, of course).

Explain the predicate stuff, too.

19.4 Nulls and Keys

Regarding ***the entity integrity rule:*** People often erroneously think this rule says that primary key values must be unique. It doesn't. It says that primary keys in base relvars can't accept nulls. Weirdnesses with this rule: It artificially distinguishes between primary and alternate keys; it artificially distinguishes between base and derived relvars (thereby violating ***The Principle of Interchangeability***).

Regarding ***foreign keys:*** Note that the apparent "need" to permit nulls in foreign keys can be avoided by appropriate database design [19.19], and such avoidance is strongly recommended.

19.5 Outer Join (a digression)

Deprecated operator! Note that:

- The result of an outer join is, in general, a mixture of two (or more) different relations with different predicates. It's **not** a relation, in general, since it "contains nulls." It would be better to stay in the relational framework and issue separate queries to obtain the separate relations that are the separate results. (Or, possibly, to extend the relational algebra to permit operators that return *sets* of relations, an idea that requires more study.)

- The operation is actually much more complicated than simple examples suggest.

- The interpretation problem raises its head again ... What do the nulls in the result of an outer join *mean?* (In fact, the only interpretation that makes any logical sense is, precisely, "value is the empty set"——see the next bullet.)

- RVAs provide a much better solution to the problem anyway. Outer join as usually understood would be **completely unnecessary** if RVAs were supported.

- Mention the possibility (and undesirability!) of defining other "outer" operations. A telling point: A version of "outer union" was added to the SQL standard with SQL:1992 and will be dropped agin with SQL:2003.

19.6 Special Values

Special values are often called *default* values, but this latter term isn't very good because it suggests something that was never intended: namely, that the value in question occurs so frequently that it might as well be the default.

The approach isn't pretty——in some ways, in fact, it's downright ugly——but it does have the virtue of staying squarely in 2VL and not wrecking the relational model. So the recommendation is to go with it, and hope that one day the researchers will come up with a *good* solution to the missing information problem. (But don't hold your breath; they've been trying for over 30 years now and don't seem to have succeeded yet.)

"There's no such thing as a null in the real world."

Stress the point that if, e.g., the "unknown" special value for HOURS_WORKED is "?", then the underlying type is *not* just INTEGER——it's a type containing integers plus "?" (loosely speaking). A nice analogy: Type TRUMPS contains five values, not four——♥, ♣, ♦, ♠, and "no trumps."

Note: Nulls advocates will argue that the special values scheme amounts to anarchy ("different users will represent missing information in different ways, thereby undermining the utility of the database as a shared resource"). And indeed there's some merit to this argument. The counterargument, of course, is that the special values approach needs to be, and can be, a **disciplined** scheme. Section 19.6 deliberately doesn't spell out much in the way of specifics, for space reasons (they're fairly obvious, anyway); see reference [19.12] for the details.

19.7 SQL Facilities

To quote: "The full implications and ramifications of SQL's null support are very complex ... For additional information, we refer you to the official standard document [4.23] or the detailed tutorial treatment in reference [4.20]." In fact, Chapter 16 of reference [4.20] provides a treatment that's meant to be not only very careful but fairly complete, too (at least as of SQL:1992). You should be aware that although SQL does support 3VL in broad outline, it also manages to make a variety of mistakes in that support. Also, you might want to develop more complete examples to illustrate the points in this section——the book contains mostly just code fragments.

Very curious behavior of type BOOLEAN!——thanks to mistaken equation of UNK and *unk*.

Horrible implications for structured and generated types (a composite value can have null components and yet not be null*). In fact, consider this example. Suppose *x* is the row value (1,*y*) and *y* IS NULL evaluates to TRUE. Then the expressions

———————

* The reason is that there's a *logical difference* between, e.g., a null row and a row all of whose components are null.

———————

x IS NULL

and

    x IS NOT NULL

both evaluate to FALSE! So x is apparently neither null nor nonnull ...

Here's another example, if you have the stomach for it (acknowledgments to Hugh Darwen for this one): Suppose x is *not* a row value, and x IS NULL evaluates to TRUE. Then

    ROW (x) IS NULL

and

    ROW (x, x) IS NULL

both evaluate to TRUE, but

    ROW (ROW (x)) IS NULL

and

    ROW (ROW (x), x) IS NULL

both evaluate to FALSE! (The problem here is that the "IS NULL" operator doesn't really mean what it says. If ROW(x,y,z) IS NULL gives TRUE, it doesn't mean the row is null——it means its *components* are null.) As Hugh says (in a private communication): "This is yet another case of incautious language design having unforeseen consequences. It was long before the ROW type generator was introduced that one was able to write, e.g., WHERE (x,y,z) IS NULL as shorthand for WHERE x IS NULL AND y IS NULL AND z IS NULL. What's more, the expression (x,y,z) was deemed to denote a row! And that's what lies behind the absurdity in question."

An extract from the text that's worth highlighting:

(Begin quote)

[You] might like to meditate on the following slightly edited extract from reference [4.20]: "Let *k2* be a new value for *K* that some user is attempting to introduce via an INSERT or UPDATE operation ... That INSERT or UPDATE will be rejected if *k2* is the same as some value for *K*, *k1* say, that already exists in the table ... What then does it mean for the two values *k1* and *k2* to be the same? It turns out that *no two* of the following three statements are equivalent:

1. *k1* and *k2* are the same for the purposes of comparison

2. *k1* and *k2* are the same for the purposes of candidate key uniqueness

3. *k1* and *k2* are the same for the purposes of duplicate elimination

Number 1 is defined in accordance with the rules of 3VL; Number 2 is defined in accordance with the rules for the UNIQUE condition; and Number 3 is defined in accordance with the definition of duplicates in Section 19.2. Suppose, for example, that *k1* and *k2* are both null; then Number 1 gives *unk*, Number 2 gives *false*, and Number 3 gives *true*.

(End quote)

Of course, the three statements *are* all equivalent in the absence of nulls. Draw your own conclusions.

Answers to Exercises

19.1 a. *unk*. b. *true*. c. *true*. d. *unk* (note the counterintuitive nature of this one). e. *false*. f. *false* (note that IS_UNK never returns *unk*). g. *false*. h. *true*.

19.2 a. *unk*. b. *unk*. c. *true*. d. *false*. e. *unk*. f. *true*. g. *false*.

19.3 Because of the following identity:

IS_UNK (*x*) ≡ MAYBE (*x* = *x*)

19.4 Because (e.g.) "MAYBE_RESTRICT *r* WHERE *p*" is the same as "*r* WHERE MAYBE(*p*)."

19.5 The four monadic operators can be defined as follows (*A* is the single operand):

A
NOT(*A*)
A OR NOT(*A*)
A AND NOT(*A*)

The 16 dyadic operators can be defined as follows (*A* and *B* are the two operands):

A OR NOT(*A*) OR *B* OR NOT(*B*)

A AND NOT(*A*) AND *B* AND NOT(*B*)
A
NOT(*A*)
B
NOT(*B*)
A OR *B*
A AND *B*
A OR NOT(*B*)
A AND NOT(*B*)
NOT(*A*) OR *B*
NOT(*A*) AND *B*
NOT(*A*) OR NOT(*B*)
NOT(*A*) AND NOT(*B*)
(NOT(*A*) OR *B*) AND (NOT(*B*) OR *A*)
(NOT(*A*) AND *B*) OR (NOT(*B*) AND *A*)

Incidentally, to see that we do not need both AND and OR, observe that, e.g.,

A OR *B* ≡ NOT (NOT (*A*) AND NOT (*B*))

19.6 See the annotation to reference [19.11].

19.7 Thanks to Exercise 19.5, it's sufficient to show that NOT and AND can both be expressed in terms of NOR:

NOT(*A*) ≡ *A*|*A*
A AND *B* ≡ (*A*|*A*)|(*B*|*B*)

Informally, NOT(*A*) is "neither *A* nor *A*"; *A* AND *B* is "neither NOT(*A*) nor NOT(*B*)." More formally, here are the truth tables. First, NOT:

A	*A*|*A*	NOT(*A*)
t	*f*	*f*
t	*t*	*f*
f	*t*	*t*
f	*f*	*t*

A	*B*	*A*|*A*	*B*|*B*	(*A*|*A*)|(*B*|*B*)	*A* AND *B*
t	*t*	*f*	*f*	*t*	*t*
t	*f*	*f*	*t*	*f*	*f*
f	*t*	*t*	*f*	*f*	*f*
f	*f*	*t*	*t*	*f*	*f*

As the exercise says, NOR is thus a "generating" operator for the whole of 2VL. Can you find an operator that performs an analogous

function for 3VL? 4VL? *n*VL? ***No answers provided*** (reference [19.20] might help).

19.8 c. For further discussion, see reference [19.5]. ***Subsidiary exercise:*** Give a relational calculus formulation for interpretation b. ***Answer:***

```
S WHERE MAYBE EXISTS SP ( SP.S# = S.S# AND
                          SP.P# = P# ( 'P2' ) )
```

19.9 We briefly describe the representation used in DB2. In DB2, a column that can accept nulls is physically represented in the stored database by two columns, the data column itself and a hidden indicator column, one byte wide, that is stored as a prefix to the actual data column. An indicator column value of binary ones indicates that the corresponding data column value is to be ignored (i.e., taken as null); an indicator column value of binary zeros indicates that the corresponding data column value is to be taken as genuine. But the indicator column is always (of course) hidden from the user.

19.10 Brief definitions:

- EXISTS (***<table exp>***) returns FALSE if the table denoted by the ***<table exp>*** is empty and TRUE otherwise.

- UNIQUE (***<table exp>***) returns TRUE if the table denoted by the ***<table exp>*** contains no two distinct rows, ***r1*** and ***r2*** say, such that the comparison ***r1*** = ***r2*** gives TRUE, and FALSE otherwise.

- ***Left*** IS DISTINCT FROM ***Right*** (where ***Left*** and ***Right*** are expressions denoting rows of the same degree, ***n*** say, and the ***i***th components of ***Left*** and ***Right*** are comparable) returns FALSE if and only if, for all ***i,*** either (a) "***Li*** = ***Ri***" gives TRUE, or (b) ***Li*** and ***Ri*** are both null; otherwise it returns TRUE.

  The operators are not primitive. Loosely:

- EXISTS(***T***) gives TRUE if SELECT COUNT(*) FROM ***T*** gives a nonzero result and FALSE otherwise.

- UNIQUE(***T***) gives TRUE if SELECT COUNT(*) FROM ***T*** = SELECT COUNT(*) FROM (SELECT DISTINCT * FROM ***T***)) gives TRUE and FALSE otherwise.

- IS DISTINCT FROM is clearly shorthand (it is effectively defined as such).

Copyright (c) 2003 C. J. Date

For some reason there's no IS NOT DISTINCT FROM operator (another example of "it would be inconsistent to fix the inconsistencies of SQL"?).

Consider this query:

```
SELECT SPJX.S# FROM SPJ AS SPJX
WHERE  SPJX.P# = P# ( 'P1' )
AND    NOT EXISTS ( SELECT * FROM SPJ AS SPJY
                    WHERE  SPJY.S# = SPJX.S#
                    AND    SPJY.P# = SPJX.P#
                    AND    SPJY.QTY = 1000 ) ;
```

("Get supplier numbers for suppliers who supply part P1 to at least one project, but only if they do not supply part P1 to any of those projects in a quantity of 1000"). Suppose relvar SPJ contains just one tuple, with supplier number S1, part number P1, project number J1, and QTY null. Then the query returns supplier S1, whereas in fact we *don't know* whether supplier S1 qualifies. The result is thus incorrect.

Consider this query:

```
SELECT UNIQUE ( SPJ.QTY )
FROM   SPJ
WHERE  SPJ.S# = S# ( 'S1' )
AND    SPJ.P# = P# ( 'P1' ) ;
```

("Is it true that there are no two projects to which supplier S1 supplies part P1 in the same quantity?"). Suppose relvar SPJ contains just two tuples, (S1,P1,J1,null) and (S1,P1,J2,null). Then the query returns TRUE, whereas in fact we *don't know* whether it should return TRUE or FALSE. The result is thus incorrect.

***** End of Chapter 19 *****

Chapter 20

T y p e I n h e r i t a n c e

Principal Sections

- Type hierarchies
- Polymorphism and substitutability
- Variables and assignments
- S by C
- Comparisons
- Operators, versions, and signatures
- Is a circle an ellipse?
- S by C revisited
- SQL facilities

General Remarks

Note the opening remarks:

> *This chapter relies heavily on material first discussed in Chapter 5. If you originally gave that chapter a "once over lightly" reading, therefore, you might want to go back and revisit it now before studying the present chapter in any depth.*

To be more specific, a clear understanding of the following is prerequisite:

- What a **type** is (reviewed in Section 20.1).

- The crucial distinction between **values** and **variables** (see Section 5.2). *Note:* Object-based discussions typically fall foul of this distinction, since they're often unclear as to whether an "object" is a value, or a variable, or both, or neither. This failure seems to be at the root of the famous (infamous?) debate as to whether, e.g., a circle is an ellipse. See Section 20.8.

- The crucial distinction between **read-only** and **update** operators (again, see Section 5.2). *Note:* The point is that read-only operators apply to *values* (possibly values that are the current values of variables), while update operators apply to *variables*.

Copyright (c) 2003 C. J. Date

- Every type has all of the following (among other things):

  - An associated **type constraint,** which defines the set of legal values of the type in question

  - At least one declared **possible representation,** together with a corresponding **selector** operator and a corresponding set of **THE_** operators (or logical equivalents of same)

  - "=" and ":=" operators

  - Certain **type testing** operators, to be discussed in Section 20.6 (these operators might be unnecessary in the absence of inheritance support); also TREAT DOWN, to be discussed in Section 20.4

  All of these bullet items except the last are also explained in Chapter 5.

The following preliminaries from Section 20.1 are also important:

- Values are typed (i.e., have **actual** "most specific" types).

- Variables are typed (i.e., have **declared** types).

- We consider **single** inheritance only in this chapter, for simplicity, though our model in fact supports multiple inheritance too.

- We consider **scalar** inheritance only in this chapter, for simplicity, though our model in fact supports tuple and relation inheritance too. Throughout the chapter, *value, variable,* and so on, thus mean **scalar** value, **scalar** variable, and so on.

- We're **not** talking about "subtables and supertables"!——we'll do that in Chapter 26.

The chapter overall is somewhat forward-looking (most database products don't provide any inheritance support, yet). In fact, at the time of writing, this book appears to be the only database textbook to include a serious discussion of type inheritance at all. (Of course, it's true that the topics are somewhat orthogonal——data doesn't have to be in a database for the concept of inheritance to apply to it——but we might say the same about the relational model, in a way.) Also, what discussions there are in other books (i.e., nondatabase books——typically books on object orientation) seem to confuse some very fundamental issues. In

this connection, note the remarks in the annotation to reference [20.2]! Note too the discussion in Chapter 26, Section 26.3, subsection "Pointers and a Good Model of Inheritance Are Incompatible," which claims, implicitly, that it's really ***objects*** and a good model of inheritance that are incompatible (since, as we'll see in Chapter 25, pointers in the shape of object IDs are a ***sine qua non*** of object orientation*). An odd state of affairs, in a way, since most of the work on inheritance seems to have been done in an object context specifically.

---

* I note in passing that this remark applies to SQL in particular, again as we'll see in Chapter 26. But it doesn't apply just to languages in which the pointers are explicit, as they are in SQL——it also applies to languages like Java where they're supposed to be completely implicit.

Be that as it may, the chapter——which can be skipped or skimmed if desired——presents a new model for inheritance, based on the proposals of reference [3.3]. It's concerned primarily with inheritance as a semantic modeling tool rather than as a software engineering tool, though we (i.e., Hugh Darwen and myself) believe the model described can meet the usual software engineering objectives——in particular, the ***code reuse*** objective——as well. ***Note:*** We justify the emphasis on the first of these two objectives by appealing to the fact that semantic modeling is more directly pertinent to the database world than software engineering is.

Our model regards ***operators*** and ***constraints*** (i.e., ***type*** constraints) as inheritable and ***structure*** as not inheritable. This position is uncontroversial with respect to operators but possibly controversial with respect to constraints and structure.* We insist on inheriting constraints because if (e.g.) a given circle violates the constraint for type ELLIPSE, then that circle isn't an ellipse! We insist on not inheriting structure because in our model there isn't any structure to inherit (structure is part of the implementation, not part of the model).

---

* Note in particular that SQL doesn't support type constraints at all, and therefore certainly doesn't support type constraint inheritance. On the other hand, it does support a form of structural inheritance. See Section 20.10 for further discussion.

Some further points to note:

- This chapter is deliberately included in this part of the book instead of Part VI in order to stress the point that the topic of inheritance, though much discussed in connection with object orientation, doesn't necessarily have anything to do with OO, and is in fact best discussed outside the OO context.

- Indeed, OO confuses the picture considerably, because (as already noted) the distinction between values and variables is absolutely crucial in this context, and that's a distinction that some people, at least, in the object world seem unwilling to make. Perhaps this fact explains why previous attempts at inheritance models haven't been very successful?

- What's more (I've already mentioned this point, but it's worth repeating and emphasizing), it's our contention that if "OO" is understood to include the notion of OIDs (see Chapter 25), then in fact it's **incompatible** with the notion of a reasonable inheritance model (i.e., one that's "faithful to reality"). In other words, OIDs and a good inheritance model can't possibly coexist, in our opinion. See the notes on Section 20.8.

- To quote Section 20.1: "The subject of type inheritance really has to do with *data in general*——it isn't limited to just *database* data in particular. For simplicity, therefore, most examples in the chapter are expressed in terms of local data (ordinary program variables, etc.) rather than database data."

20.2 Type Hierarchies

Type hierarchies are *pictures*——they're not really part of our inheritance model as such (much as "tables" are pictures, not part of the relational model as such). In other words, type hierarchies are just a convenient way of depicting certain relationships among types (supertype-subtype relationships, to be precise).

In case anyone asks: Type (e.g.) CIRCLE is not really "just circles," it's "circles at a certain position in the plane." This point notwithstanding, the book deliberately uses a rather academic example in order that the semantics can be crystal clear to everyone (?).

The subsection entitled "Terminology" is important, though fortunately straightforward. Ditto "The Disjointness Assumption," and its corollary that every value has **exactly one most specific type**.

A slightly unfortunate fact: Although we're primarily concerned with an inheritance *model,* there are certain *implementation* issues that you do need to understand in order to understand the overall concept of inheritance properly. One example: The fact that *B* is a subtype of *A* doesn't necessarily mean that the actual (hidden) representation of *B* values is the same as that of *A* values. *Implication:* Distinct implementations ("versions") of operators might be necessary under the covers. This point will become significant in the next section, among others.

The section includes this text: "So long as (a) there's at least one type and (b) there are no *cycles*——i.e., there's no sequence of types *T1, T2, T3, ..., Tn* such that *T1* is an immediate subtype of *T2, T2* is an immediate subtype of *T3, ...,* and *Tn* is an immediate subtype of *T1*——then at least one type *must* be a root type. *Note:* In fact, there *can't* be any cycles (why not?)." *Answer:* Suppose types *A* and *B* were each a subtype of the other (a cycle of length two). Then the set of values constituting *A* would be a subset of the set of values constituting *B* and *vice versa;* hence, both types would consist of exactly the same set of values. Likewise, the set of operators that applied to values of type *A* would be a subset of the set of operators that applied to values of type *B* and *vice versa* (and, of course, the set of constraints that applied to values of type *A* would be a subset of the set of constraints that applied to values of type *B* and *vice versa*). In other words, *A* and *B* would effectively be identical, except for their names, so they might as well be collapsed into a single type (in fact, we would have a violation of the model on our hands if they weren't). And, of course, an analogous argument applies to cycles of any length.

20.3 Polymorphism and Substitutability

Really the same thing. Note the need to be careful over the distinction between arguments and parameters (logical difference!). Distinguish between **overloading** and **inclusion** polymorphism; in this chapter, "polymorphism" means the latter unless otherwise stated. *Caveat:* Unfortunately, many writers use the term "overloading" to mean, specifically, inclusion polymorphism ... No wonder this subject is so confusing.

Run-time binding: CASE statements and expressions move under the covers. "Old code can invoke new code." *Note:* As a matter of fact, an implementation that did all binding at compile time (on the basis, obviously, of declared types, not most specific types) would *almost* conform to our model, because we require the semantics of operators not to change as we travel down paths in the type hierarchy (see Section 20.7). The reason I say "almost" here, however, is that compile-time binding clearly won't work——in fact, it's impossible——for *dummy types*. Dummy types aren't discussed in detail in the book, however; see reference [3.3] for further details.

Substitutability——more precisely, *value* substitutability——is **the** justification for inheritance!

20.4 Variables and Assignments

Important message: **Values retain their most specific type on assignment to variables of less specific declared type** (type conversion does *not* occur on such assignment). Hence, a variable of declared type *T* can have a value whose most specific type is **any subtype** of *T*. So we also need to be careful over the difference between the *declared* type of a given variable and the *actual* (most specific) type of the current value of that variable (another important logical difference). Formal model of a variable, and more generally of an expression: *DT, MST, v* components.

If operator *Op* is defined to have a result of declared type *T,* then the actual result of an invocation of *Op* can be of **any subtype** of type *T*. *Note:* We deliberately do not drag in the (in our experience, rather confusing and unhelpful) terms and concepts *result covariance* and *argument contravariance*. "Result contravariance" is just an obvious consequence of substitutability (what's more, the term doesn't seem to capture the essence of the phenomenon properly). And we don't believe in "argument contravariance" at all, for reasons articulated in reference [3.3].

TREAT DOWN (important); possibility of run-time type errors (in this context and nowhere else).

20.5 S by C

Basic idea: If variable E of declared type ELLIPSE is updated in such a way that now THE_A(E) = THE_B(E), then *MST*(E) is now CIRCLE. After all, human beings know that an ellipse with equal semiaxes is really a circle, so the system ought to know the same

thing——otherwise the model can hardly be said to be "faithful to reality" or "a good model of reality."

Caveat: Most inheritance models do ***not*** support S by C; in fact, some writers are on record as arguing that an inheritance model should explicitly **not** support it (see, e.g., reference [20.12]). By contrast, we believe an inheritance model is useful as "a model of reality" only if it does support S by C (and we believe we know how to implement it efficiently, too).

Be warned that the term "S by C" (or something very close to it, anyway) is used elsewhere in the literature with a very different meaning; see, e.g., reference [20.14], where it's used to refer to what would better be called just ***type constraint enforcement***. Here's the definition from that reference:

(Begin quote)

"***Specialization via constraints*** happens whenever the following is permitted:

B subtype_of ***A*** and ***T*** subtype_of ***S*** and
f(...***b:T,***...) returns ***r:R*** in Ops(***B***) and
f(...***b:S,***...) returns ***r:R*** in Ops(***A***)

That is, specialization via constraints occurs whenever the operation redefinition on a subtype constrains one of the arguments to be from a smaller value set than the corresponding operation on the supertype."

(End quote)

This definition lacks somewhat in clarity, it might be felt.

Anyway, S by C (in our sense) implies, very specifically, that a selector invocation might have to return a value of more specific type than the specified "target" type. In other words, ***the implementation code for S by C is embedded in selector code***. (That implementation code can probably be provided automatically, too.)

Explain ***G by C*** as well.

20.6 Comparisons

Self-explanatory——though the implications for join etc. sometimes come as a bit of a surprise.

Copyright (c) 2003 C. J. Date

Explain IS_*T* and the new relational operator *R*:IS_*T*(*A*). *Note:* Generalized versions of these operators are defined in reference [3.3].

20.7 Operators, Versions, and Signatures

Much confusion in the literature over different kinds of signatures! Need to distinguish **specification signature** (just one of these) *vs*. **version signatures** (many) *vs*. **invocation signatures** (also many). More logical differences here, in fact ...

Changing operator semantics as we travel down the type hierarchy is, regrettably, possible but (we believe) nonsense. Arguments in favor are (we believe) based on a confusion between inclusion and overloading polymorphism and smack of "the implementation tail wagging the model dog" [3.3]. Changing semantics is illegal in our model.

Discuss **union types** briefly (or at least mention them). *Note:* Some proposals——e.g., ODMG [25.11]——use union types as a way of providing *type generator* functionality. E.g., RELATION might be a union type in such a system (with generic operators JOIN, UNION, and so forth), and every specific relation type would then be a proper subtype of that union type. We don't care for this approach ourselves, because we certainly don't want our support for type generators to rely on support for type inheritance. What's more, the approach seems to imply that specific——i.e., explicitly specialized——implementation code must be provided for each specific join, each specific union, etc., etc.: surely not a very desirable state of affairs? How can it be justified?

The section shows an explicit implementation of the MOVE operator (read-only version) that moves circles instead of ellipses, and then remarks that "there's little point in defining such an explicit [implementation] in this particular example (why, exactly?)." *Answer:* Because S by C will take care of the problem!

20.8 Is a Circle an Ellipse?

IMPORTANT!——albeit self-explanatory, more or less.* But you should be aware that this is another, and major, area where we depart from "classical" inheritance models. To be specific, it's here that the value *vs*. variable and read-only *vs*. update operator distinctions come into play. Other approaches don't make these distinctions; they thus allow operators (update as well as read-only operators) to be inherited indiscriminately——with the result that they have to support "noncircular circles" and similar

nonsenses, and they can't support type constraints at all! (SQL is very unfortunately a case in point here. See Section 20.10.)

---

* I don't much care for "advertisements for myself," but I do think you should take a look at reference [20.6] if you propose to teach the material of this section.

The section includes the following text: "[Let] type ELLIPSE have another immediate subtype NONCIRCLE; let the constraint *a* > *b* apply to noncircles; and consider an assignment to THE_A for a noncircle that, if accepted, would set *a* equal to *b*. What would be an appropriate semantic redefinition for that assignment? Exactly what side effect would be appropriate?" ***No answer provided!***——the questions are rhetorical, as should be obvious.

20.9 S by C Revisited

This section begins by criticizing the common example of colored circles as a subtype of circles. Note that there can't be more instances (meaning more ***values***) of a subtype than of any supertype of that subtype, yet there are clearly more colored circles than there are circles. And colored circles can't be obtained from circles via S by C, either. Note the remark to the effect that "COLORED_CIRCLE is a subtype of CIRCLE to exactly the same extent that it is a subtype of COLOR (which is to say, not at all)." In my experience, most students find this point telling.

Discussion of this example leads to the position that **S by C is the *only* conceptually valid means of defining subtypes**——the exact opposite of the position articulated in reference [20.12] and subscribed to by much of the object world.

20.10 SQL Facilities

Extremely unorthogonal!——basically single inheritance only, for "structured types" only.* (Multiple inheritance might be added in SQL:2003.)

---

* As the book says: SQL has no explicit inheritance support for generated types, no explicit support for multiple inheritance, and no inheritance support at all for built-in types or DISTINCT types. But it does have some very limited *implicit* support for inheritance of generated types and for multiple inheritance.

Explain the SQL analog of circles and ellipses. Inheritance not of constraints and (read-only) operators but structure and (all) operators; explain implications! Functions, procedures, and methods. Observers, mutators, and constructors. No type constraints; this omission is staggering but a necessary consequence of SQL's inheritance model (?). Do *not* get into details of reference types or subtables and supertables here (we'll cover them in Chapter 26, after we've discussed OO in Chapter 25).

Explain **delegation**——it's pragmatically important, but it's not inheritance (in our opinion).

References and Bibliography

We repeat the opening paragraph from this section:

(Begin quote)

For interest, we state here without further elaboration the sole major changes required to [our single] inheritance model ... in order to support multiple inheritance. First, we relax the disjointness assumption by requiring only that *root* types must be disjoint. Second, we replace the definition of "most specific type" by the following requirement: Every set of types *T1, T2, ..., Tn* ($n \geq 0$) must have a common subtype *T'* such that a given value is of each of the types *T1, T2, ..., Tn* if and only if it is of type *T'*. See reference [3.3] for a detailed discussion of these points, also of the extensions required to support tuple and relation inheritance.

(End quote)

Reference [20.1] describes a commercial implementation of the inheritance model as described in the body of the chapter. Reference [20.10] is a good example of what happens if the value *vs.* variable and read-only *vs.* update operator distinctions are ignored; unfortunately, it very much reflects what SQL does (see Section 20.10). Reference [20.12] is interesting as an example of how the object world thinks about inheritance, though we caution

you that (as indicated earlier) we reject almost all of its stated positions.

Answers to Exercises

20.1 Some of the following definitions elaborate slightly on those given in the body of the chapter.

- **Code reuse** means a given program might be usable on data that is of a type that didn't even exist when the program was written.

- **Delegation** means the responsibility for implementing certain operators associated with a given type is "delegated" to the type of some component of that type's representation. It's related to operator overloading.

- Let *T'* be a proper subtype of *T*, let *V* be a variable of declared type some supertype of *T*, and let *MST*(*V*) be *T'*. The term **generalization by constraint** refers to the fact that, after assignment to *V*, *MST*(*V*) will be generalized (revised upward) to *T* if *v*(*V*) satisfies the type constraint for *T* but not for any proper subtype of *T*.

- Type *T'* is an **immediate subtype** of type *T* if it's a subtype of *T* and there's no type *T''* that's both a proper supertype of *T'* and a proper subtype of *T*.

- **Inheritance:** If type *T'* is a subtype of type *T*, then all constraints and read-only operators that apply to values of type *T* are inherited by values of type *T'* (because values of type *T' are* values of type *T*). Update operators that apply to variables of declared type *T* might or might not be inherited by variables of declared type *T'*.

- A **leaf type** is a type with no proper subtype.

- The term **polymorphism** refers to the possibility that a given operator can take arguments of different types on different invocations. Several different kinds of polymorphism exist: *inclusion* polymorphism (the principal kind of interest for the present chapter); *overloading* polymorphism (where distinct operators happen to have the same name); *generic* polymorphism (e.g., the relational project operator is generic in the sense that it applies generically to relations of *all possible* relation types); and so on.

- Type *T'* is a **proper subtype** of type *T* if it's a subtype of *T* and *T'* and *T* are distinct.

- A **root type** is a type with no proper supertype.

- **Run-time binding** is the process of determining at run time which particular implementation version of a polymorphic operator to execute in response to a particular invocation.

- The term **signature** means, loosely, the combination of the *name* of some operator and the *types of the operands* to the operator in question (note, however, that different writers and different languages ascribe slightly different meanings to the term; e.g., the *result* type is sometimes regarded as part of the signature, and so too are operand and result *names*). It is important to distinguish **specification signature** *vs*. **version signatures** *vs*. **invocation signatures** (see Section 20.7).

- Let *T'* be a proper subtype of *T*, let *V* be a variable of declared type some supertype of *T*, and let *MST*(*V*) be *T*. The term **specialization by constraint** refers to the fact that, after assignment to *V*, *MST*(*V*) will be specialized (revised downward) to *T'* if *v*(*V*) satisfies the type constraint for *T'* but not for any proper subtype of *T'*.

- The term **substitutability** (of values) refers to the fact that wherever the system expects a value of type *T*, we can always substitute a value of type *T'* instead, where *T'* is a subtype of *T*. The term "substitutability of *variables*" refers to the fact that wherever the system expects a variable of declared type *T*, we *might* be able to substitute a variable of declared type *T'* instead, where (again) *T'* is a subtype of *T*.

- A **union type** (also known as an "abstract" or "noninstantiable" type, or sometimes just as an "interface") is a type that isn't the most specific type of any value at all. Such a type provides a way of specifying operators that apply to several different **regular** types, all of them proper subtypes of the union type in question.

20.2 Consider the expression TREAT_DOWN_AS_*T*(*X*), where *X* is an expression. *MST*(*X*) must be a subtype of *T* (this is a run-time check). If this condition is satisfied, the result *Y* has *DT*(*Y*) equal to *T*, *MST*(*Y*) equal to *MST*(*X*), and *v*(*Y*) equal to *v*(*X*).

20.3 *No answer provided*.

20.4 The least specific type of *any* value of *any* of the types shown in Fig. 20.1 is PLANE_FIGURE, of course.

20.5 22 (this count includes the empty hierarchy).

20.6 Since all rectangles are centered on the origin, a rectangle ABCD can be uniquely identified by any two adjacent vertices, say A and B. To pin matters down more precisely (and using Cartesian coordinates), let A be the point (*xa,ya*) and B the point (*xb,yb*); then C is (*-xa,-ya*) and D is (*-xb,-yb*). Since A, B, C, and clearly lie on a circle with center the origin, we clearly must have $xa^2 + ya^2 = xb^2 + yb^2$. Thus, we can define type RECTANGLE as follows:

```
TYPE RECTANGLE IS PLANE_FIGURE
     POSSREP { A POINT, B POINT
     CONSTRAINT THE_X ( A ) ** 2 + THE_Y ( A ) ** 2 =
                THE_X ( B ) ** 2 + THE_Y ( B ) ** 2 } ;
```

Such a rectangle is a square if and only if the vertex B = (*xb,yb*) = (*ya,-xa*). Thus, we can define type SQUARE as follows:

```
TYPE SQUARE IS RECTANGLE
     CONSTRAINT THE_X ( THE_B ( RECTANGLE ) ) =
                THE_Y ( THE_A ( RECTANGLE ) ) AND
                THE_Y ( THE_B ( RECTANGLE ) ) =
              - THE_X ( THE_A ( RECTANGLE )
     POSSREP { A = THE_A ( RECTANGLE ) } ;
```

Note: For a detailed explanation of the syntax of the POSSREP and CONSTRAINT specifications (which as you can see is different in the two cases shown here), see reference [3.3].

For interest, we give another solution involving a polar possrep instead:

```
TYPE RECTANGLE IS PLANE_FIGURE
     POSSREP { A POINT, B POINT
     CONSTRAINT THE_R ( A ) = THE_R ( B ) } ;

TYPE SQUARE IS RECTANGLE
     CONSTRAINT ABS ( THE_θ ( THE_A ( RECTANGLE ) ) -
                      THE_θ ( THE_B ( RECTANGLE ) ) ) = Π / 2
     POSSREP { A = THE_A ( RECTANGLE ) } ;
```

20.7 The operators defined below are update operators specifically.

```
OPERATOR ROTATE ( T RECTANGLE ) UPDATES T
   VERSION ROTATE_RECTANGLE ;
```

```
          THE_X ( THE_A ( T ) )  :=  - THE_X ( THE_B ( T ) ) ,
          THE_Y ( THE_A ( T ) )  :=    THE_Y ( THE_B ( T ) ) ,
          THE_X ( THE_B ( T ) )  :=  - THE_X ( THE_A ( T ) ) ,
          THE_Y ( THE_B ( T ) )  :=    THE_Y ( THE_A ( T ) ) ;
    END OPERATOR ;

    OPERATOR ROTATE ( S SQUARE ) UPDATES S
       VERSION ROTATE_SQUARE ;
    END OPERATOR ;
```

Note that the ROTATE_SQUARE version is (reasonably enough) essentially just a "no-op."

Polar analogs:

```
    OPERATOR ROTATE ( T RECTANGLE ) UPDATES T
       VERSION ROTATE_RECTANGLE ;
       THE_θ ( THE_A ( T ) )  :=  THE_θ ( THE_A ( T ) ) + Π / 2 ,
       THE_θ ( THE_B ( T ) )  :=  THE_θ ( THE_B ( T ) ) + Π / 2 ;
    END OPERATOR ;

    OPERATOR ROTATE ( S SQUARE ) UPDATES S
       VERSION ROTATE_SQUARE ;
    END OPERATOR ;
```

As a subsidiary exercise, define some read-only analogs of those operators. *Answer:*

```
    OPERATOR ROTATE ( T RECTANGLE ) RETURNS RECTANGLE
       VERSION ROTATE_RECTANGLE ;
       RETURN RECTANGLE ( POINT ( - THE_X ( THE_B ( T ) ),
                                    THE_Y ( THE_B ( T ) ) ),
                          POINT ( - THE_X ( THE_A ( T ) ),
                                    THE_Y ( THE_A ( T ) ) ) ) ;
    END OPERATOR ;

    OPERATOR ROTATE ( S SQUARE ) RETURNS SQUARE
       VERSION ROTATE_SQUARE ;
       RETURN S ;
    END OPERATOR ;
```

Polar analogs:

```
    OPERATOR ROTATE ( T RECTANGLE ) RETURNS RECTANGLE
       VERSION ROTATE_RECTANGLE ;
       RETURN
          RECTANGLE ( POINT ( THE_R ( THE_A ( T ) ),
                              THE_θ ( THE_A ( T ) ) + Π / 2 ),
                      POINT ( THE_R ( THE_B ( T ) ),
                              THE_θ ( THE_B ( T ) ) + Π / 2 ) ) ;
```

```
END OPERATOR ;

OPERATOR ROTATE ( S SQUARE ) RETURNS SQUARE
   VERSION ROTATE_SQUARE ;
   RETURN S ;
END OPERATOR ;
```

20.8

a. The specified expression will fail on a compile-time type error, because THE_R requires an argument of type CIRCLE and the declared type of A is ELLIPSE, not CIRCLE. (Of course, if the compile-time type check were not done, we would get a *run-time* type error instead as soon as we encountered a tuple in which the A value was just an ellipse and not a circle.)

b. The specified expression is valid, but it yields a relation with the same heading as R, not one in which the declared type of attribute A is CIRCLE instead of ELLIPSE.

20.9 The expression is shorthand for an expression of the form

```
( ( EXTEND ( R ) ADD ( TREAT_DOWN_AS_T ( A ) ) AS A' )
                                  { ALL BUT A } ) RENAME A' AS A
```

(where *A'* is an arbitrary name not already appearing as an attribute name in the result of evaluating *R*).

20.10 The expression is shorthand for an expression of the form

```
( R WHERE IS_T ( A ) ) TREAT_DOWN_AS_T ( A )
```

Moreover, this latter expression is itself shorthand for a longer one, as we saw in the answer to Exercise 20.9.

20.11 *No answer provided.*

20.12 *No answer provided.*

***** End of Chapter 20 *****

Copyright (c) 2003 C. J. Date

Chapter 21

Distributed Databases

Principal Sections

- Some preliminaries
- The twelve objectives
- Problems of distributed systems
- Client/server systems
- DBMS independence
- SQL facilities

General Remarks

Distributed databases can arise in two distinct ways:

1. The database was always intended to be unified from a logical point of view, and was designed that way, but is physically distributed for performance or similar reasons.

2. The database is an after-the-fact unification of a set of previously existing databases at a set of distinct sites.

Both cases are important. More recently, however, the emphasis (for a variety of obvious pragmatic reasons) has been on Case 2 rather than Case 1. Case 2 is often referred to as "federated" or (this term is less widespread) "multi-database" systems; the term "middleware" is relevant here, too. Possibly mention the Web. Data integration is a hot topic!——see, e.g., reference [21.9].

It should be clear that federated systems are likely to run into nasty problems of *semantic mismatch* and the like (see Section 21.6), though the problems of Case 1 are hardly trivial either.

Distributed systems as parallel processing systems?

Client/server systems as a simple special case of distributed systems in general.

This is mostly implementation stuff, not model stuff! The chapter can be skipped or skimmed if desired.

21.2 Some Preliminaries

Copyright (c) 2003 C. J. Date

The **strict homogeneity** assumption effectively means we're dealing with Case 1, until further notice. The assumption is adopted primarily for pedagogic reasons (it simplifies the presentation); we consider what happens when it's relaxed in Section 21.6.

The fundamental principle of distributed database (which ideally ought to apply to both Case 1 and Case 2):

> *To the user, a distributed system should look exactly like a* ***non****distributed system.*

The twelve objectives (useful as an organizing principle for discussion but *not* necessarily hard and fast requirements, and not necessarily all equally important):

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

21.3 The Twelve Objectives

Mostly self-explanatory. A few notes on individual objectives are appropriate, however.

Local autonomy: Obviously desirable, but not 100 percent achievable. The following list of cases where it isn't is taken from the annotation to reference [21.13]:

- Individual fragments of a fragmented relvar can't normally be accessed directly, not even from the site at which they're stored.

- Individual copies of a replicated relvar (or fragment) can't normally be accessed directly, not even from the site at which they're stored. (Actually, certain of today's so-called "replication products" do allow such direct access, but they're using the term "replication" in a rather different sense. See Section 21.4, subsection "Update Propagation." See also Chapter 22.)

- Let *P* be the primary copy of some replicated relvar (or fragment) *R,* and let *P* be stored at site *X*. Then every site that accesses *R* is dependent on site *X,* even if another copy of *R* is in fact stored at the site in question.

- *(Important!)* A relvar that participates in a multi-site integrity constraint can't be accessed for update purposes within the local context of the site at which it's stored, but only within the context of the distributed database in which the constraint is defined. Note the implications for defining, e.g., a foreign key constraint on existing data that spans sites! (probably can't do things like cascade delete)——especially in a "federated" system.

- A site that's acting as a participant in a two-phase commit process must abide by the decision (i.e., commit or rollback) of the corresponding coordinator site.

No reliance on a central site: One implication is that we want *distributed solutions* to various problems (e.g., lock management, deadlock detection).

Continuous operation: Define *reliability* and *availability*. No planned shutdowns! Note in particular the implication that Release *N*+1 of the DBMS at site *A* must be able to work with Release *N* at site *B* (upgrading the DBMS release level simultaneously at every site is infeasible).

Location independence: An extension of the familiar concept of (physical) *data* independence; in fact, every objective in the list that has "independence" in its name is an extension of physical data independence.

Fragmentation independence: Note the parallels with view processing. The section includes the following text: "[Relvar] EMP as perceived by the user might be regarded, loosely, as a [union] *view* of the underlying fragments N_EMP and L_EMP ... *Exercise:* Consider what is involved on the part of the optimizer in dealing with the request EMP WHERE SALARY > 40K." *Answer:* First, it transforms the user's original request into the following:

```
( N_EMP UNION L_EMP ) WHERE SALARY > 40K
```

This expression can then be transformed further into:

```
( N_EMP WHERE SALARY > 40K )
  UNION
( L_EMP WHERE SALARY > 40K )
```

Copyright (c) 2003 C. J. Date

The system can thus execute the two restrictions at the appropriate sites and then form the union of the results.

Replication independence: Replication with replication independence is a special case of controlled redundancy (see Chapter 1). Mention update propagation but defer detailed discussion.

Distributed query processing: Self-explanatory.

Distributed transaction management: Note the term "agent"——it doesn't seem to be used much in the literature, but *some* term is clearly needed for the concept.

Hardware, operating system, and network independence: Self-explanatory.

DBMS independence: Forward pointer to Section 21.6.

21.4 Problems of Distributed Systems

All of these "problems"——as already noted——ideally require *distributed solutions*.

Query processing: Stress the importance of *optimizability* (as opposed to optimization *per se*). Distributed systems really must be relational if they're ever going to perform (unless, perhaps, "the seams show," meaning performance is back, partly, in the hands of the user).

Catalog management: Naming is crucial. The R* scheme is elegant and worth describing, but you can substitute discussion of some alternative (commercial?) scheme if you prefer. *Question:* If TABLE is the catalog relvar that lists all named relvars, what does the query SELECT * FROM TABLE do (in any particular system you happen to be familiar with)? What *should* it do? These questions might be useful as the basis of a class discussion.

Update propagation: Describe the primary copy scheme, plus any more sophisticated scheme you might care to (but are such schemes actually implemented anywhere?). Explain the difference between "true" replication as described here and the typical "replication" product as supported by today's commercial DBMS vendors, which is probably asynchronous and might not provide replication independence. Refer backward to *snapshots* (Chapter 10) and forward to *data warehouses* (Chapter 22).

Recovery control: Explain **two-phase commit** very carefully——the basic version, plus any refinements you think are worth discussing (*presumed commit* and *presumed abort,* at least). Consider the possibility of failures at various points in the overall process. It's impossible to make the process 100 percent resilient to any conceivable kind of failure. (So what do real systems do? *Answer:* They sometimes force a rollback when a commit would have been OK.)

Concurrency control: Discuss the primary copy scheme and the possibility of global deadlock.

21.5 Client/Server Systems

Be clear on the fact that the term "client/server" refers primarily to an *architecture,* or logical division of responsibilities; the **client** is the application and the **server** is the DBMS. Usually but not necessarily at different sites (on different hardware platforms). Mention the term "two-tier system."

Set-level processing is important! So too might be *stored procedures* and RPC (= remote procedure call).

Mention RDA and DRDA, perhaps also the four DRDA levels of functionality [21.22] (remote request, remote unit of work, distributed unit of work, distributed request).

21.6 DBMS Independence

First, discuss *gateways* (*aka,* more specifically, *point-to-point* gateways, and more recently *wrappers*). Serious technical problems, even in this limited case!——especially if the target system is nonrelational. (The reason for mentioning this obvious fact is that's there a huge amount of hype out there regarding the capabilities of this kind of system, and that hype needs to be challenged. As the chapter says, it's obviously possible to provide some useful functionality, but it's *not* possible to do a 100 percent job.)

Next, discuss *data access middleware* (the "federated database" stuff). An increasingly important kind of product, but (again) there's no magic ... Certain seams are going to show, despite what the vendor might say.

A useful way to think about a data access middleware product (though not the way such products are usually characterized in the literature) is as follows: From the point of view of an

individual client, *it looks like a regular DBMS.*[*] However, the data is stored, mostly, not at the middleware site, but rather at any number of other sites behind the scenes, under the control of a variety of other DBMSs (or even file managers). In other words, the middleware product uses the combination of those other DBMSs and/or file managers as its own *storage manager* (and coordinates their operation, of course).

---

[*] In the case of DataJoiner, at least, it *is* a DBMS (among other things). Why would you buy DB2 when you can buy DataJoiner instead? (The question is hypothetical, or rhetorical, but the point is that not all technical questions have technical answers! The answer to this particular question probably has more to do with IBM's marketing and pricing strategies than it does with technical issues.)

21.7 SQL Facilities

Explain client/server capabilities——CONNECT, DISCONNECT, SET CONNECTION (not in too much detail). By the way, note the syntax: CONNECT *TO* but not DISCONNECT *FROM* (this point isn't mentioned in the book). You could elaborate on SQL/PSM's stored procedure support if you like, but it's complicated (see reference [4.20]).

Answers to Exercises

21.1 Location independence means users can behave (at least from a logical standpoint) as if the data were all stored at their own local site. **Fragmentation independence** means users can behave (at least from a logical standpoint) as if the data weren't fragmented. **Replication independence** means users can behave (at least from a logical standpoint) as if the data weren't replicated.

21.2 Here are some of the reasons:

- Ease of data fragmentation
- Ease of data reconstruction
- Set-level operations
- Optimizability

21.3 See Section 21.2.

21.4 See Section 21.4.

21.5 See Section 21.4.

21.6 *No answer provided.*

21.7 *No answer provided.*

21.8 *No answer provided.*

***** End of Chapter 21 *****

Chapter 22

D e c i s i o n S u p p o r t

Principal Sections

- Aspects of decision support
- DB design for decision support
- Data preparation
- Data warehouses and data marts
- OLAP
- Data mining
- SQL facilities

General Remarks

David McGoveran was the original author of this chapter.

The term *decision support* covers a multitude of sins! (After all, classical query processing could certainly be regarded as decision support, of a kind; so too could traditional transaction processing, perhaps with a bit of a stretch.) This chapter begins by giving some historical perspective, then concentrates on the currently fashionable notions of (a) "data warehouses," "data marts," and so forth, and (b) "online analytical processing" (OLAP). It also includes with a brief look at the application of statistical techniques to discover patterns in very large volumes of data——*data mining* (a comparatively new field, made possible by the combined availability of cheap computer storage and fast computer processing). It concludes with a sketch of the pertinent features of SQL.

The chapter is, primarily, a high-level overview of what by now is a large subject in its own right. An important quote from Section 22.1: "We remark immediately that one thing [these areas] all have in common is that good logical design principles are rarely followed in any of them! The practice of decision support is, regrettably, not as scientific as it might be; often, in fact, it's quite *ad hoc*. In particular, it tends to be driven by physical considerations much more than by logical ones——indeed, it tends to blur the logical *vs*. physical distinction considerably." *Caveat lector*.

We use SQL, not **Tutorial D**, as the basis for examples; we use the "fuzzy" terminology of rows, columns, and tables in place of tuples, attributes, and relation values and variables (relvars);

Copyright (c) 2003 C. J. Date

we use *logical schema* and *physical schema* in place of *conceptual schema* and *internal schema*.

The chapter can be skipped or skimmed if desired.

22.2 Aspects of Decision Support

Key point: The database is primarily read-only (except for periodic *load* or *refresh* operations). Also:

- Columns tend to be used in combination.
- Integrity in general is not a concern; the data is assumed to be correct when first loaded and isn't subsequently updated. (These facts don't mean we don't have to declare integrity constraints, though!——see the next section.)
- Keys often include a temporal component.
- The database tends to be large.
- The database tends to be heavily indexed.
- The database often involves various kinds of *controlled redundancy* (including "summary tables" as well as straight data replication).

Decision support queries tend to be quite *complex*. Here are some of the kinds of complexities that can arise:

- Boolean expression complexity
- Join complexity
- Function complexity
- Analytical complexity

All of the foregoing factors lead to a strong emphasis on *designing for performance*. Of course, this fact should affect only the physical design of the database, not the logical design, but (as previously noted) vendors and users both typically fail to distinguish properly between the two ... *segue* into the next section.

22.3 DB Design for Decision Support

Self-explanatory. Observe in particular:

- The treatment of composite columns

- The fact that integrity constraints need to be considered and stated, even though the database is read-only

- The issues concerning "temporal keys" (forward pointer to Chapter 23)

Note especially the remarks concerning physical design and the subsection on common design errors (especially with respect to "star schemas"——forward pointer to Section 22.5).

22.4 Data Preparation

Also self-explanatory. Note the discussion of **extract** in particular (if the section is covered at all——but it could easily be skipped).

22.5 Data Warehouses and Data Marts

Note first that these terms aren't very precisely defined! Loosely, however, a data mart is (a copy of) some "hot subset" of the data warehouse. Discuss the desirability of separating decision support and operational processing. There are arguments (in fact, they seem to be warming up a little these days) in favor of integrating them, too.

Describe **dimensional schemas** ... **star schemas** ... **fact** and **dimension tables**. Explain "star join." What's the difference between a star schema and a normal schema? This question is hard to answer with *simple* examples, because a simple star schema can look very similar (even identical) to a good relational design. In fact, however, there are several problems with the star schema approach in general:

- It's *ad hoc* (based on intuition, not principle).

- Star schemas tend to be physical, not logical.

- Sometimes information is lost.

- The fact table often contains several different types of facts.

- The dimension tables can become nonuniform, too.

- The dimension tables are often less than fully normalized.

Note: One reviewer of the previous edition said: "[This section] is critical of the star schema [approach] but proposes no alternative." Actually, the section isn't so much critical of star schemas as such (how could it be, without a precise definition of the concept?); rather, it's critical of the fact that, very often, what people call a "star schema" is simply a bad logical design. And, of course, the section does implicitly propose an alternative: namely, *good* logical design (i.e., design done in accordance with well-established relational design principles, as described in Chapters 12 and 13).

22.6 OLAP

Analytical processing always implies *data aggregation,* usually according to many different groupings. In classical relational languages (and in SQL too, prior to SQL:1999), each individual query involves at most one grouping (perhaps implicit) and produces just one table as its result; hence, *n* distinct groupings require *n* distinct queries, producing *n* distinct results. It thus seems worth trying to find a way:

a. Of requesting several levels of aggregation in a single query, and thereby

b. Offering the implementation the opportunity to compute all of those aggregations more efficiently (i.e., in a single pass).

Such considerations are the motivation behind the *GROUPING SETS, ROLLUP,* and *CUBE* options on the GROUP BY clause found in certain SQL implementations and also (since SQL:1999) in the SQL standard as well.

Bundling several queries into one statement might be a good idea, but bundling the results into one table isn't (basically because the result isn't a relation). *What's the predicate?* (Always a good question to ask!)

Explain *crosstabs*. Note that crosstabs aren't a very good way to display a result involving more than two dimensions——and the more dimensions there are, the worse it gets (see Exercise 22.9).

Describe *multi-dimensional databases* (relate to crosstabs). ROLAP *vs*. MOLAP. Sparse arrays (point out that these are an artifact of the representation, not a "feature"!).

Please criticize the position that "relations are two-dimensional." There's massive confusion out there in the

marketplace on this extremely simple point. A couple of genuine (bad) quotes in this regard:

- "When you're well trained in relational modeling, you begin to believe the world is two-dimensional. You think you can get anything into the rows and columns of a table" [Douglas Barry, Executive Director, ODMG].

- "There is simply no way to mask the complexities involved in assembling two-dimensional data into a multi-dimensional form" [Richard Finkelstein].

22.7 Data Mining

Data mining is a huge subject in its own right (there are whole books devoted to the topic). The purpose of this section is only to scratch the surface of the subject, nothing more. Probably sufficient just to go through the simple SALES example. Explain the terms *population, support level, confidence level.*

The purpose of the final paragraph in this section is simply to make the student aware of the *names* of certain techniques and (perhaps) to give the faintest of ideas of what each of those techniques can do. It's deliberately not meant to be fully understandable.

22.8 SQL Facilities

GROUPING SETS, ROLLUP, and CUBE were included in the SQL:1999 standard as originally published; other facilities were added the following year in the "OLAP amendment" [22.21]. But this stuff isn't *database,* it's *statistics*——and the details don't belong in a database book, in my opinion. (They might belong in an *SQL* book, of course.) Thus, the intent of this section is merely to give a sense of the scope of that "OLAP amendment," nothing more.

References and Bibliography

Note the introductory remark:

(Begin quote)

The "views" mentioned in the titles of references [22.3-22.5], [22.10], [22.12], [22.16], [22.25], [22.28], [22.30], and [22.35] are not views but snapshots. Annotation to those references talks in terms of snapshots, not views.

(End quote)

Answers to Exercises

22.1 To quote from Section 22.5: "Operational systems usually have strict performance requirements, predictable workloads, small units of work, and high utilization. By contrast, decision support systems typically have varying performance requirements, unpredictable workloads, large units of work, and erratic utilization. These differences can make it very difficult to combine operational and decision support processing within a single system——conflicts arise over capacity planning, resource management, and system performance tuning, among other things. For such reasons, operational system administrators are usually reluctant to allow decision support activities on their systems; hence the familiar dual-system approach."

22.2 To quote from Section 22.4: "The data must be ***extracted*** (from various sources), ***cleansed, transformed*** and ***consolidated, loaded*** into the decision support database, and then periodically ***refreshed***."

22.3 Controlled redundancy is redundancy that's known to and managed by the DBMS (involving, in particular, automatic update propagation). Such redundancies might or might not be visible to the user. **Uncontrolled** redundancy is (of course) redundancy that isn't controlled in the foregoing sense and must therefore be managed by the user.

Indexes and the transaction log are both examples of controlled redundancy; so too is replication in the sense of Chapter 21. Maintaining separate detail and summary information "by hand" is an example of uncontrolled redundancy.

Redundancy is important for decision support because it can make query formulation simpler and query execution faster. Such redundancy is obviously better if it's ***controlled,*** however, because (as with declarative support for queries and the like) "controlled" means the system does the work, while "uncontrolled" means the user does the work.

22.4 *No answer provided.*

22.5 *No answer provided.*

22.6 *No answer provided.*

22.7 In ROLAP, the user sees the data in relational form and issues relational-style queries. In MOLAP, the user sees the data

as a multi-dimensional array and issues array-style queries (more or less).

22.8 There are eight (= 2^3) possible groupings for each hierarchy, so the total number of possibilities is 8^4 = 4,096. As a subsidiary exercise, you might like to consider what's involved in using SQL to obtain all of these summarizations. ***No further answer provided*** (the question is rhetorical, somewhat).

22.9 With respect to the SQL queries, we show the GROUP BY clauses only:

a. GROUP BY GROUPING SETS ((S#,P#), (P#,J#), (J#,S#))

b. GROUP BY GROUPING SETS (J#, (J#,P#), ())

c. The trap is that the query is ambiguous——the term (e.g.) "rolled up along the supplier dimension" has many possible meanings. However, one possible interpretation of the requirement will lead to a GROUP BY clause looking like this:

   GROUP BY ROLLUP (S#), ROLLUP (P#)

d. GROUP BY CUBE (S#, P#)

We omit the SQL result tables. As for the crosstabs, it should be clear that crosstabs aren't a very good way to display a result that involves more than two dimensions (and the more dimensions there are, the worse it gets). For example, one such crosstab——corresponding to GROUP BY S#, P#, J#——might look like this (in part):

	P1				P2				...
	J1	J2	J3	...	J1	J2	J3	...	...
S1	200	0	0	...	0	0	0	...	...
S2	0	0	0	...	0	0	0	...	...
S3	0	0	0	...	0	0	0	...	...
S4	0	0	0	...	0	0	0	...	...
S5	0	200	0	...	0	0	0	...	...
..	...	...	...	...	...	...	...	...	...

In a nutshell: The headings are clumsy, and the arrays are sparse.

22.10 *No answer provided.*

22.11 Perhaps. Debate!

Copyright (c) 2003 C. J. Date

22.12 *No answer provided.*

***** End of Chapter 22 *****

Chapter 23

T e m p o r a l D a t a b a s e s

Principal Sections

- What's the problem?
- Intervals
- Packing and unpacking relations
- Generalizing the relational operators
- DB design
- Integrity constraints

General Remarks

The problem of how to handle temporal data has always been important but in recent times has become much more so, thanks in part to the increasing interest in data warehouses (recall the references to temporal issues in Chapter 22). And a sizable section of the research community has been at work for some years on a temporal proposal called *TSQL2*. It's therefore as well to say right up front that Chapter 23 is *not* based on TSQL2, because TSQL2 suffers from certain serious technical flaws, the most fundamental of which is that it violates *The Information Principle* and is thus not truly relational. (Just as an aside, I note that most of the work reported in the literature on temporal matters describes itself as relational but really isn't. It took me a while to realize this fact! Once I'd done so, however, I found it very helpful in understanding just what it was that the various researchers were proposing. Thus, I seriously suggest that you take this observation on board——the observation, that is, that "relational" in the literature doesn't always mean relational. I think you might find it helpful too.)

To say it again, Chapter 23 is not based on TSQL2. Instead, it's based on *sound relational principles* (what else?). It describes an approach, originally due to Nikos Lorentzos and elaborated in reference [23.4], that, we hope and believe,* will soon be of more than just theoretical significance. Like Chapter 20 on type inheritance, therefore, this chapter is forward-looking, in the sense that it describes not how systems work today but, rather, how we think they ought to work in the future. (Incidentally, it turns out that the inheritance model described in Chapter 20 is relevant to the temporal proposals of reference [23.4], though the aspects in question are beyond the scope of the present book. FYI, they have to do with what's called

Copyright (c) 2003 C. J. Date

"granularity"——i.e., the ability to perceive the very same period of time, say the year 2003, sometimes as a sequence of 12 months and sometimes as a sequence of 365 days.)

———————

* We = Hugh Darwen, Nikos, and myself.

———————

The chapter can be skipped if desired but probably shouldn't just be skimmed (individual topics are tightly interwoven). If you do cover it, you should probably have a copy of reference [23.4] to hand, for reference purposes if nothing else. That book covers a lot of material that there wasn't room for in Chapter 23. Of course, it also goes into much more detail on the topics there *was* room for.

Section 23.1 is longer than most if not all of the other chapter introductions. It lays some important groundwork. Explain the following concepts carefully:

- *Timestamped propositions,* corresponding (external) *predicates,* and the careful and precise interpretations we give to **since** and **during;**
- Intuitive idea of *points* and *intervals* (closed-closed style);
- "Beginning of time" and "end of time" points and the notion of *successor* (and predecessor);
- A little more formally: The interval with begin point *b* and end point *e* is the sequence of all points *p* such that $b \leq p \leq e$ (where "<" means "earlier than").

Explain the first (i.e., nontemporal) version of the running example——"suppliers and shipments," a drastically simplified and reinterpreted version of suppliers and parts. *Note:* This database looks much too simple to be useful, but in fact it's sufficient to illustrate the vast majority of ideas introduced in the chapter. What's more, if we made the database more "realistic" by, say, not discarding supplier city information, we'd get into some difficulties (having to do with database design) that I don't want to discuss——actually, we're not *equipped* to discuss——at this early juncture.

Explain the primary and foreign key constraints, also Queries A and B. Sketch the plan of the rest of the chapter. *Note:* FYI,

here are some of the topics *not* included in this chapter but covered in reference [23.4]: general queries; updates; "valid time *vs.* transaction time"; implementation and optimization; "cyclic point types"; granularity and scale; continuous point types; and (important!) syntactic shorthands for many of the foregoing.

Although the chapter is called "Temporal Databases" and concentrates on temporal issues, the ideas are actually of wider applicability: As we've already seen, the basic construct we're dealing with is *intervals,* and intervals don't necessarily have to be temporal (as we'll see). Note too that:

- A lot of the terminology we'll be using——"packed form," U_key, and much more——is nonstandard (but then there *are* no standard terms for many of these concepts).

- There's no "SQL Facilities" section in this chapter. A while back, however, the standards committee was seriously considering an "SQL/Temporal" component, and it was leaning heavily toward an approach very similar to that described in this chapter. We can expect interest to revive in this subject at some point in the future, so we might eventually see some SQL support for what we're going to be talking about. In fact, we hope our work in reference [23.4] will exert some positive influence on the committee! We have a chance to do it right, since none of the vendors has done it wrong (yet).

- This is not a closed subject!——certain interesting research issues remain. Some of those issues are touched on here and there in passing.

One final introductory point (paraphrased remark from Section 23.3): "It's worth pointing out that the INTERVAL type generator is the sole construct introduced in this chapter that isn't just shorthand. Our approach to temporal databases——unlike others that have been described in the literature——thus involves no changes at all to the classical relational model (although it does involve certain *generalizations,* as we'll see in Sections 23.5 and 23.7)."* In other words, almost everything we're going to be talking about——the new operators and other new constructs——can be expressed (albeit only very longwindedly, in many cases) in terms of features already available in a complete relational language such as **Tutorial D**. This point is important, and it illustrates what is in my opinion the "right" way to do language design in general.

———————

Copyright (c) 2003 C. J. Date

* The generalizations in question are generalizations of joins, unions, keys, and other familiar relational concepts——but even these generalizations are really still just shorthand.

_____

23.2 What's the Problem?

This section extends suppliers and shipments to:

a. A "semitemporal" version, and then

b. A "fully temporal" version, without using intervals as such (the fully temporal version with intervals as such comes in the next section).

It traces what happens to (a) the simple primary and foreign key constraints, and (b) Queries A and B, as we move from the nontemporal version of the database through these two temporal versions. ***Note the very careful wording of the various extended versions of Queries A and B.*** The overall message of the section is that life gets complicated, fast, if we stay with traditional data types and traditional operators only. We need something new!——though (as far as possible) we'd like that "something new" to be basically just a set of carefully thought-out shorthands.

Regarding the ***semitemporal*** ("since") version, note:

a. The revised predicates;

b. The additional constraint XST1;

c. The fact that an appropriately extended version of Query B can't be done (because the database is only **semi**temporalized, so far). We need to keep ***historical records*** showing which suppliers were able to supply which parts when.

Regarding the first ***fully temporal*** ("from-to") version, note:

a. The further revisions to the predicates, plus additional semantic assumptions;

b. The further revisions to the key constraints and the additional constraints S_FROM_TO_OK, SP_FROM_TO_OK, XFT1 ("no overlapping and no abutting"), XFT2, and especially XFT3 (complicated!);

c. The fact that the queries are now staggeringly complex (we don't even attempt to give formulations).

Note too:

a. The first mention of combining tuples;

b. The assumption that today is day 10;

c. The questions arising from that assumption (forward pointer to Section 23.6).

The closing sentence from this section: "In a nutshell, then, the problem of temporal data is that it quickly leads to constraints and queries (not to mention updates, which are beyond the scope of this chapter) that are unreasonably complex to express: unreasonably complex, that is, unless the system provides some appropriate shorthands, which commercial DBMSs currently don't."

23.3 Intervals

Intervals are the abstraction we need. Show the fully temporal version of the running example with intervals. Note the predicates. Explain:

- Point types: total ordering, FIRST_*T*, LAST_*T*, NEXT_*T*, PRIOR_*T* (NEXT_*T* is the ***successor function***). ***Note:*** The need for those "_*T*" qualifiers is explained in reference [23.4] but not in Chapter 23. I wouldn't get into it unless asked (and even then I'd take it offline).

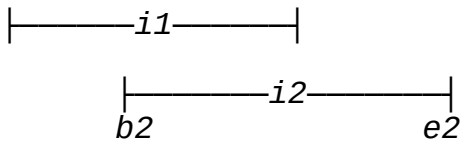
- The INTERVAL type generator and interval types: generic possrep, generic operators, generic constraints; selectors and "THE_" operators BEGIN and END.

Mention examples of nontemporal point types and intervals. Discuss other operators on points and intervals: IS_NEXT_*T*, IS_PRIOR_*T*, MAX, MIN, ε and its inverse, COUNT, POINT FROM, PRE and POST. Allen's operators (our version!). UNION, INTERSECT, MINUS (you might want to leave MINUS as an exercise, it's a bit tricky and possibly not as important as the other two).

Show some sample queries.

Note: The text suggests that students might like to try drawing some pictures to illustrate Allen's operators. This is the kind of thing I had in mind:

- ***OVERLAPS:*** (*i1* OVERLAPS *i2*) ≡ (*b1* ≤ *e2* AND *b2* ≤ *e1*)



Pictures like this can be useful as an aid to memory, especially since the operator names aren't always all that self-explanatory.

23.4 Packing and Unpacking Relations

First explain EXPAND and COLLAPSE (unary-relation versions only; other versions are described in reference [23.4]). Crucial notion of *equivalence* for such unary relations. The operators are shorthand, though they might not look like it! State without attempting to justify the nullary-relation versions too.

Very carefully explain PACK and UNPACK on one interval attribute. Work through Query A, leading up to the point where it can be formulated thus:

```
PACK SP_DURING { S#, DURING } ON DURING
```

Shorthand! Mention analogy with grouping, *but* (to quote) "while *r* GROUP {*A*} ... is guaranteed to return a result with exactly one tuple for each distinct value of *B* (where *B* is all of the attributes of *r* apart from *A*), PACK *r* ON *A* might return a result with several tuples for any given value of *B*" (explain).

Note that the definition of PACK——and UNPACK too, when we get there——relies, at least conceptually, on support for relation-valued attributes (so it's not "just shorthand" if such support isn't available).

Work through Query B, probably less exhaustively:

```
PACK
  ( ( UNPACK S_DURING { S#, DURING } ON DURING )
    MINUS
    ( UNPACK SP_DURING { S#, DURING } ON DURING ) )
ON DURING
```

Shorthand! Mention analogy with grouping (etc.). *Note:* This is still not the final formulation of Query B——we'll come back to it in the next section.

Do some further examples (those in the book or some of your own). Include at least one nontemporal one.

Quote from the end of the section: "There is much, much more to the PACK and UNPACK operators than we have room for in this

chapter. Detailed discussions can be found in reference [23.4]; here we simply list without proof or further commentary some of the most important points." You should at least mention each of the four bulleted points:

- Packing and unpacking on no attributes
- Unpacking on two or more attributes (straightforward)
- Packing on two or more attributes (tricky)
- Extended notion of equivalence

23.5 Generalizing the Relational Operators

Use the Query B discussion by way of motivation, leading up to U_MINUS. By the way: That discussion suggests, but never quite states explicitly, that we don't really want the implementation to do unpacks. Very true! UNPACK is a *conceptually* important component of our approach, but we want to avoid actual unpackings if possible——and we can (see reference [23.4], Appendix A).

We use "arrowhead brackets" (* and *) for clarity in our exposition, but parentheses would probably be OK in a real concrete syntax.

Run through U_UNION, U_INTERSECT, U_JOIN, U_project. See reference [23.4] for the rest (U_SUMMARIZE in particular, which is also quite useful; U_restrict etc. are not all that useful but are included for completeness). All just more shorthand. Show "final" formulations of Queries A and B:

```
USING DURING * SP_DURING { S#, DURING } *

USING DURING * S_DURING { S#, DURING }
               MINUS
                SP_DURING { S#, DURING } *
```

Relational comparisons: Explain "U_=" in particular. Relate to the previously discussed notion of equivalence.

The regular relational operations revisited: Regular operations are just special cases of their generalized counterparts! A most pleasing and important state of affairs.

23.6 DB Design

"Special issues arise in connection with the design of temporal databases." True ... First let's revise our running example:

- Drop shipments entirely.

- Reinstate supplier name, status, and city information.

Show preferred design immediately, noting that (a) S_SINCE in that design is not the same as S_SINCE in Section 23.2 and (b) S_DURING in that design is not the same as S_DURING in Section 23.3. State the predicates. Introduce the terms **horizontal decomposition** and **vertical decomposition**.

Regarding ***horizontal decomposition:*** Explain the logical difference between historical and current information. Explain why "timestamping tuples" doesn't work (the timestamps timestamp too much, and updating is ugly).

Regarding ***vertical decomposition:*** Without it, the timestamps timestamp too much, and updating is ***very*** ugly. ***Segue*** into **sixth normal form** (6NF) ... 6NF basically just means irreducibility, but the reduction (or decomposition) operator is not plain old projection any longer but ***generalized*** projection. Likewise, the recomposition operator is ***generalized*** join. The definition of 6NF relies on ***generalized*** JDs (all classical JDs are generalized JDs, but some generalized JDs aren't classical JDs). All 6NF relvars are in 5NF, but some 5NF relvars aren't in 6NF. ***Note:*** Because 6NF relies on ***generalized*** JDs, its existence doesn't undermine statements elsewhere to the effect that 5NF is "the final normal form"! 5NF ***is*** the final normal form, so long as we limit ourselves to classical projection and join. On the other hand, it's utterly reasonable to think of 6NF as another step in the same old progression from 1NF to 2NF to ... to 5NF.

Return to horizontal decomposition to introduce (and debunk, or——more politely——deconstruct) a notion mentioned in the literature that goes by the general name of "the moving point ***now***" ... Point out some of the logical nonsenses that this notion leads us into; perhaps draw a parallel between NOW and NULL ("from the people who brought you NULL, we now present ..."). [*] "[It's] precisely an argument in favor of horizontal decomposition that NOW isn't needed."

———————

[*] Indeed, NOW is really a ***variable,*** and the notion of ***values*** containing ***variables*** makes no sense.

———————

Note: We *could* have "during" relvars only (i.e., no horizontal decomposition), if we really wanted——but only if we have no objection to our database telling lies. ("Telling lies" = using "the end of time" to mean "until further notice"). Not a good idea!——the propositions in the database are supposed to be true ones.

23.7 Integrity Constraints

Now we have the conceptual apparatus we need to deal with the issue of constraints* (recall how complicated constraints were when we first examined them, without benefit of the interval abstraction, associated operators, and so forth). Focus on S_STATUS_DURING until further notice ... We examine three general problems: *redundancy, circumlocution, contradiction*.

_____

* Also with the issues of query and update, but we're not going to get into details of those here.

_____

Regarding *redundancy:* KEY is inadequate; tuples need to be packed. Need to enforce the following constraint: If at any given time relvar S_STATUS_DURING contains two distinct tuples that are identical except for their DURING values *i1* and *i2,* then *i1* OVERLAPS *i2* must be false.

Regarding *circumlocution:* Again, KEY is inadequate; tuples need to be packed. Need to enforce the following constraint: If at any given time relvar S_STATUS_DURING contains two distinct tuples that are identical except for their DURING values *i1* and *i2,* then *i1* MEETS *i2* must be false.

Since OVERLAPS OR MEETS ≡ MERGES, we can fix both of the foregoing problems via PACKED ON (explain). *Note:* PACKED ON is just a constraint——it's *not* intended to cause automatic repacking after updates (if anybody asks). Automatic repacking, if supported, would be a "compensating action" (like cascade delete). We don't say that's wrong, but we don't regard it as fundamental, and foundations are what we're interested in here (as always). We do believe the possibility of compensating actions would need careful study, however, if seriously proposed (they've led to some complexity in the past).

Regarding *contradiction:* PACKED ON and KEY are still inadequate, even together. Need to enforce the following constraint: If at any given time relvar S_STATUS_DURING contains two tuples that have the same S# value but differ on their STATUS value, then their DURING values *i1* and *i2* must be such that *i1* OVERLAPS *i2* is false. WHEN / THEN does the trick (explain).

Explain *U_keys*. Shorthand! Regular keys a special case! Foreign keys can be treated analogously, but we omit the details (just give an example). *Note:* Although the chapter doesn't discuss updating, you might like to mention that we have "U_" versions of INSERT, DELETE, and UPDATE too ... and the regular versions are all degenerate special cases of those U_ versions. (We ought really to have "U_assign" too, but reference [23.4] doesn't explicitly discuss such a possibility.)

Integrity is much more than just keys ... To quote: "Reference [23.4] presents a careful and detailed analysis of the overall problem; to be specific, it considers, in in very general terms, a set of nine requirements that we might want a typical temporal database like the suppliers-and-shipments database to satisfy." Show the nine requirements; perhaps justify intuitively; *don't* try to get into details, unless you have a *lot* of time available! It's tricky stuff. In any case, the details aren't in the book.

References and Bibliography

Note the following: "Rather than giving what could easily be a very lengthy list of references here, we draw your attention to the extensive bibliography in reference [23.4], *q.v.*" Note too reference [23.3], which analyzes and criticizes TSQL2, if you want to be prepared for questions on that topic. (You might well be asked such questions, since TSQL2 has received a certain amount of emphasis in the literature. In fact, there's a book available on how to deal with the time dimension in the absence of system support, and that book is heavily based on the TSQL2 ideas:

- Richard T. Snodgrass: *Developing Time-Oriented Database Applications in SQL*. San Francisco, Calif.: Morgan Kaufmann (2000).

Answers to Exercises

23.1 A **time quantum** (also known as a *chronon*) is the smallest time unit the system is capable of representing. A **time point** is the time unit that is relevant for some particular purpose. **Granularity** is the "size" or duration of the applicable time

points, or equivalently the "size" or duration of the gap between adjacent points.

23.2 See Section 23.3.

23.3 The following edited extract from reference [23.4] goes part way to answering this exercise.

(Begin quote)

Replacing the pair of attributes FROM and TO by the single attribute DURING in each of the two relvars brings with it a number of immediate advantages. Here are some of them:

- It avoids the problem of having to make an arbitrary choice as to which of two candidate keys should be regarded as primary. For example, relvar S_FROM_TO had two candidate keys, {S#,FROM} and {S#,TO}, but relvar S_DURING has just one, {S#,DURING}, which we can therefore designate as "primary" (if we wish) without any undesirable arbitrariness. Similarly, relvar SP_FROM_TO also had two candidate keys but relvar SP_DURING has just one, {S#,P#,DURING}, which again we can designate as "primary" if we wish.

- It also avoids the problem of having to decide whether the FROM-TO intervals in the previous version of the database are to be interpreted as closed or open with respect to FROM and TO. Previously, those intervals were implicitly taken to be closed with respect to both FROM and TO. But now, e.g., [*d04*:*d10*], [*d04*:*d11*), (*d03*:*d10*], and (*d03*:*d11*) are four distinct possible representations of the very same interval, and we have no need to know which, if any, is the actual physical representation. (See reference [23.4] for further explanation of the terms "open" and "closed" as used here.)

- Yet another advantage is that integrity constraints to guard against the absurdity of a FROM-TO pair appearing in which the TO value is less than the FROM value are no longer necessary, because the constraint "FROM ≤ TO" is implicit in the very notion of an interval type. That is, constraints of the form "FROM ≤ TO" are effectively replaced by a *generic* constraint that implicitly applies to each and every individual interval type.

- Suppose relations *r1* and *r2* were both to include distinct FROM and TO attributes (albeit with different names in each case), instead of a single DURING attribute, and suppose we were to join *r1* and *r2* to produce *r3*. Then *r3* would contain two FROM-TO attribute pairs, and it would be the user's

responsibility, not the system's, to match up the FROMs and TOs appropriately. Clearly, this problem (though admittedly psychological, not logical, in nature) will only get worse as the number of joins increases, and it has the potential to give rise to serious human errors. What's more, the difficulties would be compounded if we were to discard some of the FROMs and/or TOs by means of projections. Such problems don't arise——or, at least, are much less severe——with DURING attributes.

(End quote)

In addition, of course, treating intervals as values in their own right is what enables us to define all of the new operators and other constructs that we need to formulate queries, constraints, and so forth in an intellectually manageable way.

23.4 INTERVAL_INTEGER ([BEGIN(*i*) - COUNT(*i*) :
END(*i*) + COUNT(*i*)])

Evaluation of this expression will fail at run time if either of the following expressions evaluates to TRUE:

- FIRST_INTEGER() + COUNT(*i*) > BEGIN(*i*)
- LAST_INTEGER() - COUNT(*i*) < END(*i*)

23.5 INTERVAL_INTEGER ([BEGIN(*i*) + COUNT(*i*) / 3 :
END(*i*) - COUNT(*i*) / 3])

23.6 INTERVAL_INTEGER
[MIN (MIN (BEGIN(*i1*), BEGIN(*i2*)), BEGIN(*i3*)) :
MAX (MAX (END(*i1*), END(*i2*)), END(*i3*))]

We've assumed for definiteness that INTEGER is the underlying point type. Note that the following expression——

i1 UNION *i2* UNION *i3*

——might not work, because UNION isn't necessarily defined for every pair of intervals taken from the given three.

23.7 Yes, if the expression on the right side is defined; otherwise no. Here are three examples (simplified notation):

- *a* = [2:6], *b* = [4:9]; *a* INTERSECT *b* = [4:6], *a* MINUS (*a* MINUS *b*) = [2:6] MINUS [2:3] = [4:6].
- *a* = [4:6], *b* = [2:6]; *a* INTERSECT *b* = [4:6], *a* MINUS *b* (and hence *a* MINUS (*a* MINUS *b*) undefined.

- *a* = [4:6], *b* = [8:9]; *a* INTERSECT *b* undefined, *a* MINUS *b* = [4:6], *a* MINUS (*a* MINUS *b*) undefined.

23.8 (a) Suppose there's a total ordering on part numbers, say P1 < P2 < P3 (etc.). Then the following relation might be interpreted to mean that certain suppliers were able to supply certain ranges of parts during certain intervals of time:

S#	PARTS	DURING
S1	[P1:P3]	[*d01*:*d04*]
S1	[P2:P4]	[*d07*:*d08*]
S1	[P5:P6]	[*d09*:*d09*]
S2	[P1:P1]	[*d08*:*d09*]
S2	[P1:P2]	[*d08*:*d08*]
S2	[P3:P4]	[*d07*:*d08*]
S3	[P2:P4]	[*d01*:*d04*]
S3	[P3:P5]	[*d01*:*d04*]
S3	[P2:P4]	[*d05*:*d06*]
S3	[P2:P4]	[*d06*:*d09*]
S4	[P3:P4]	[*d05*:*d08*]

(b) The following relation might be interpreted to mean that certain ranges of suppliers were able to supply certain ranges of parts during certain intervals of time:

SUPPLIERS	PARTS	DURING
[S1:S2]	[P2:P3]	[*d03*:*d03*]
[S1:S2]	[P2:P2]	[*d04*:*d04*]
[S1:S3]	[P3:P3]	[*d04*:*d04*]
[S2:S3]	[P3:P4]	[*d05*:*d05*]
[S2:S3]	[P4:P4]	[*d04*:*d04*]

(c) See (b) above.

23.9 The first assertion is valid, the second isn't. For proof, see reference [23.4].

23.10
```
WITH ( FEDERAL_GOVT RENAME DURING AS FD ) AS FG ,
     ( STATE_GOVT   RENAME DURING AS SD ) AS SG ,
     ( FG JOIN SG ) AS T1 ,
     ( T1 WHERE FD OVERLAPS SD ) AS T2 ,
     ( EXTEND T2 ADD ( FD INTERSECT SD ) AS DURING ) AS T3 :
  T3 { ALL BUT FD, SD }
```

23.11 The following example is taken from reference [23.4]. We're given a relvar INFLATION representing the inflation rate for a certain country during certain specified time intervals. A sample value is given below; it shows that the inflation rate was 18 percent for the first three months of the year, went up to 20 percent for the next three months, stayed at 20 again for the *next* three months (but went up to 25 percent in month 7), ..., and averaged out at 20 percent for the year as a whole.

DURING	PERCENTAGE
[*m01*:*m03*]	18
[*m04*:*m06*]	20
[*m07*:*m09*]	20
[*m07*:*m07*]	25
.....	..
[*m01*:*m12*]	20

The constraint PACKED ON DURING mustn't be specified for this relvar because (in terms of the sample value shown above) such a constraint would cause the three tuples with PERCENTAGE = 20 to be "packed" into one, and we'd lose the information that the inflation rate for months 4-6 and months 7-9 (as well as for the year overall) was 20 percent.

23.12 Let *r1* and *r2* be as follows:

r1

A
[*d01*:*d05*]
[*d08*:*d10*]

r2

A
[*d02*:*d02*]
[*d04*:*d09*]

Then the cardinality of the relation produced by USING A * *r1* INTERSECT *r2* * is three:

A
[*d02*:*d02*]
[*d04*:*d05*]
[*d08*:*d09*]

23.13 We need to show that

UNPACK T6 ON *A* ≡ (UNPACK *r1* ON *A*) JOIN (UNPACK *r2* ON *A*)

Assume first that *r1* and *r2* each have just the one attribute *A*. Then T6 consists of every possible intersection of a DURING value from *r1* and a DURING value from *r2*. It follows that the unpacked form of T6 consists (loosely) of every unit interval that's contained in at least one of those intersections (and therefore in some DURING value in *r1* and in some DURING value in *r2*). It's clear that the join of the unpacked forms of *r1* and *r2* consists (loosely) of the very same unit intervals.

Now assume that *r1* and *r2* have some additional attributes, *B* say. If we partition each relation on the basis of *B* values, we can apply an argument analogous to that given above to each pair of partitions, one each from *r1* and *r2*.

"Confirm also that if *r1* and *r2* are both initially packed on *A*, then the final PACK step is unnecessary": *No answer provided*.

23.14 See Section 23.6. The answer to the second part of the exercise is yes (again, see Section 23.6).

23.15 *No answer provided*.

23.16

a.
```
( ( SUMMARIZE SP_SINCE BY S#
    ADD ( COUNT AS CT, MIN ( SINCE ) AS MS ) )
  WHERE CT > 1 ) { S#, MS }
```

b. Can't be done. We can get the supplier numbers but not the dates:

```
( ( SUMMARIZE SP_SINCE BY S# ADD COUNT AS CT )
  WHERE CT = 1 ) { S# }
```

23.17 See Section 23.7.

23.18 See Section 23.7.

***** End of Chapter 23 *****

Copyright (c) 2003 C. J. Date

Chapter 24

Logic-Based Databases

Principal Sections

- Overview
- Propositional calculus
- Predicate calculus
- A proof-theoretic view of databases
- Deductive database systems
- Recursive query processing

General Remarks

No "SQL Facilities" section in this chapter, for obvious reasons.

The following remarks from Section 24.1 should be pretty much self-explanatory:

(Begin quote)

In the mid 1980s or so, a significant trend began to emerge in the database research community toward **database systems that are based on logic**. Expressions such as *logic database, inferential DBMS, expert DBMS, deductive DBMS, knowledge base, knowledge base management system (KBMS), logic as a data model, recursive query processing,* etc., etc., began to appear in the research literature. However, it isn't always easy to relate such terms and the ideas they represent to familiar database terms and concepts, nor to understand the motivation underlying the research from a traditional database perspective; in other words, there's a clear need for an explanation of all of this activity in terms of conventional database ideas and principles. This chapter is an attempt to meet that need. Our aim is to explain what logic-based systems are all about from the viewpoint of someone who's familiar with traditional database technology but perhaps not so much with logic as such. As each new idea from logic is introduced, therefore, we'll explain it in conventional database terms, where possible or appropriate. (Of course, we've discussed certain ideas from logic in this book already, especially in our description of relational calculus in Chapter 8. Relational calculus is directly based on logic. However, there's more to logic-based systems than just the relational calculus, as we'll see.)

Copyright (c) 2003 C. J. Date

(End quote)

There's still no consensus on whether logic-based systems as such will ever make it into the mainstream, but certainly a lot of research is still going on, as evidenced by the annual SIGMOD proceedings, VLDB proceedings, etc. (On the other hand, most of the ***functionality*** provided by logic-based systems **is** finding its way into the SQL standard and/or mainstream products in some shape or form; recursive queries are a case in point.)

Note the summarized definitions of terms in Section 24.8 (in particular, explain the concept of "logic as a data model").

The chapter can be skipped if desired. In particular, it probably ***should*** be skipped if Chapter 8 (on relational calculus) was skipped earlier.

24.2 Overview

Explain **model-theoretic** *vs.* **proof-theoretic** perceptions (in outline). Discuss ***deductive axioms*** (rules by which, given certain facts, we're able to deduce additional facts). Of course, deductive axioms are really just views by another name (and facts are really just tuples, as should already be clear from discussions in numerous earlier chapters).

24.3 Propositional Calculus

A tutorial for database people. Basically straightforward. Describe the **resolution** technique carefully.

24.4 Predicate Calculus

Again, a tutorial for database people. Note the big difference between propositional and predicate calculus: **Predicate calculus allows formulas to contain (logic) variables and quantifiers**. E.g., "Supplier S1 supplies part ***p***" and "Some supplier ***s*** supplies part ***p***" aren't legal formulas in the propositional calculus, but they ***are*** legal in the predicate calculus. Thus, predicate calculus provides a basis for expressing queries such as "Which parts are supplied by supplier S1?" or "Get suppliers who supply some part."

Review **free** and **bound** variable references and **open** and **closed WFFs** (all previously explained in Chapter 8). Explain **interpretations** and **models:**

- An *interpretation* of a set of WFFs is the combination of a universe of discourse, plus the mapping of individual constants to objects in that universe, plus the defined meanings for the predicates and functions with respect to that universe.

- A *model* of a set of WFFs is an interpretation for which all WFFs in the set are *true*.

  Describe **clausal form** and **resolution and unification**.

24.5 A Proof-Theoretic View of Databases

A *clause* is an expression of the form

A1 AND *A2* AND ... AND *Am* =* *B1* OR *B2* OR ... OR *Bn*

where the *A*'s and *B*'s are all terms of the form

r (*x1, x2, ..., xt*)

(where *r* is a predicate and *x1, x2, ..., xt* are the arguments to that predicate). Two important special cases:

1. *m* = 0, *n* = 1: The clause is basically just

   r (*x1, x2, ..., xt*)

   for some predicate *r* and some set of arguments *x1, x2, ..., xt*. If the *x*'s are all constants, the clause represents a **ground axiom**——i.e., it is a statement (a closed WFF, in fact) that is unequivocally *true*. In database terms, such a statement corresponds to a tuple of some relvar *R*.

2. *m* > 0, *n* = 1: The clause takes the form

   A1 AND *A2* AND ... AND *Am* =* *B*

   which can be regarded as a **deductive axiom;** it gives a definition of the predicate on the right side in terms of those on the left side. Alternatively, it can be regarded as an **integrity constraint**.

Explain (properly this time!) the difference between **model-** and **proof-theoretic** perceptions. Summarize the axioms for a given database (proof-theoretic view). Introduce the term **extensional database**.

24.6 Deductive Database Systems

The axioms mentioned in the previous section don't mention *integrity constraints*——because (in the proof-theoretic view) adding constraints converts the system into a **deductive** system. A deductive system is one that supports the proof-theoretic view, and in particular one that can deduce additional facts from the given facts in the extensional database by applying specified **deductive axioms** or **rules of inference**. The deductive axioms, plus integrity constraints, constitute the **intensional database**.

Sketch the "deductive" version of suppliers and parts (including the recursive axioms needed to represent part structure). Explain **Datalog** briefly ("the entire deductive database can be regarded as a Datalog program") and mention possible extensions to that language.

24.7 Recursive Query Processing

As the title indicates, this section is concerned with (simple) implementation techniques, not with how to formulate recursive queries (that's already been covered). Note that many more sophisticated techniques are described in the references. Briefly discuss:

- Unification and resolution
- Naïve evaluation
- Seminaïve evaluation
- Static filtering
- Other algorithms as desired (the so-called "magic" techniques [24.16-24.19] might be worth some discussion, but stress that they aren't applicable *only* to "logic-based systems"——they can be used in conventional systems too, as the annotation to reference [18.22] explains)

Answers to Exercises

24.1 a. Valid. b. Valid. c. Not valid.

24.2 In the following, *a*, *b*, and *c* are Skolem constants and *f* is a Skolem function with two arguments.

a. *p* (*x*, *y*) =* *q* (*x*, *f* (*x*, *y*))

b. *p* (*a*, *b*) =* *q* (*a*, *z*)

c. *p* (*a*, *b*) =* *q* (*a*, *c*)

24.3 We consider part a. only. We have:

1. WOMAN (Eve)
2. PARENT (Eve, Cain)
3. MOTHER (*x*, *y*) *= PARENT (*x*, *y*) AND WOMAN (*x*)

Rewrite 3. to eliminate "*=":

4. MOTHER (*x*, *y*) OR NOT PARENT (*x*, *y*) OR NOT WOMAN (*x*)

Negate the conclusion and adopt as a premise:

5. NOT MOTHER (Eve, Cain)

Substitute Eve for *x* and Cain for *y* in line 4 and resolve with line 5:

6. NOT PARENT (Eve, Cain) OR NOT WOMAN (Eve)

Resolve 2. and 6.:

7. NOT WOMAN (Eve)

Resolve 1. and 7.: We obtain the empty set of clauses [].

24.4 An **interpretation** of a set of WFFs is the combination of a universe of discourse, plus the mapping of individual constants to objects in that universe, plus the defined meanings for the predicates and functions with respect to that universe. A **model** of a set of WFFs is an interpretation for which all WFFs in the set are true.

24.5 *No answer provided.*

24.6 In accordance with our usual practice, we have numbered the following solutions as 24.6.*n*, where 7.*n* is the number of the original exercise in Chapter 7. As in the body of the chapter, we write 300 as a convenient shorthand for QTY(300), etc.

24.6.13 ? *= J (*j*, *jn*, *jc*)

24.6.14 ? *= J (*j*, *jn*, London)

24.6.15 RES (*s*) *= SPJ (*s*, *p*, J1)

Copyright (c) 2003 C. J. Date

```
        ? *= RES ( s )
```

24.6.16
```
        ? *=  SPJ ( s, p, j, q ) AND 300 ≤ q AND q ≤ 750
```

24.6.17
```
        RES ( pl, pc ) *= P ( p, pn, pl, w, pc )
        ? *= RES ( pl, pc )
```

24.6.18
```
        RES ( s, p, j ) *= S ( s, sn, st, c ) AND
                            P ( p, pn, pl, w, c ) AND
                           J ( j, jn, c )
        ? *= RES ( s, p, j )
```

24.6.19-24.6.20 Can't be done without negation.

24.6.21
```
        RES ( p ) *= SPJ ( s, p, j, q ) AND
                      S ( s, sn, st, London )
        ? *= RES ( p )
```

24.6.22
```
        RES ( p ) *= SPJ ( s, p, j, q ) AND
                     S ( s, sn, st, London ) AND
                     J ( j, jn, London )
        ? *= RES ( p )
```

24.6.23
```
        RES ( c1, c2 ) *= SPJ ( s, p, j, q ) AND
                           S ( s, sn, st, c1 ) AND
                          J ( j, jn, c2 )
        ? *= RES ( c1, c2 )
```

24.6.24
```
        RES ( p ) *= SPJ ( s, p, j, q ) AND
                      S ( s, sn, st, c ) AND
                     J ( j, jn, c )
        ? *= RES ( p )
```

24.6.25 Can't be done without negation.

24.6.26
```
        RES ( p1, p2 ) *= SPJ ( s, p1, j1, q1 ) AND
                          SPJ ( s, p2, j2, q2 )
        ? *= RES ( p1, p2 )
```

24.6.27-24.6.30 Can't be done without grouping and aggregation.

24.6.31
```
        RES ( jn ) *= J ( j, jn, jc ) AND
                      SPJ ( S1, p, j, q )
        ? *= RES ( jn )
```

24.6.32
```
        RES ( pl ) *= P ( p, pn, pl, w, pc ) AND
                      SPJ ( S1, p, j, q )
        ? *= RES ( pl )
```

24.6.33
```
        RES ( p ) *= P ( p, pn, pl, w, pc ) AND
```

```
                    SPJ ( s, p, j, q ) AND
                  J ( j, jn, London )
        ? *= RES ( p )
```

24.6.34
```
RES ( j ) *= SPJ ( s, p, j, q ) AND
             SPJ ( S1, p, j2, q2 )
        ? *= RES ( j )
```

24.6.35
```
RES ( s ) *= SPJ ( s, p, j, q ) AND
               SPJ ( s2, p, j2, q2 ) AND
             SPJ ( s2, p2, j3, q3 ) AND
               P ( p2, pn, Red, w, c )
        ? *= RES ( s )
```

24.6.36
```
RES ( s ) *= S ( s, sn, st, c ) AND
             S ( S1, sn1, st1, c1 ) AND st < st1
        ? *= RES ( s )
```

24.6.37-24.6.39 Can't be done without grouping and aggregation.

24.6.40-24.6.44 Can't be done without negation.

24.6.45
```
RES ( c ) *= S ( s, sn, st, c )
RES ( c ) *= P ( p, pn, pl, w, c )
RES ( c ) *= J ( j, jn, c )
        ? *= RES ( c )
```

24.6.46
```
RES ( p ) *= SPJ ( s, p, j, q ) AND
               S ( s, sn, st, London )
RES ( p ) *= SPJ ( s, p, j, q ) AND
             J ( j, jn, London )
        ? *= RES ( p )
```

24.6.47-24.6.48 Can't be done without negation.

24.6.49-24.6.50 Can't be done without grouping.

24.7 We show the constraints as conventional implications instead of in the "backward" Datalog style.

a.
```
CITY ( London )
CITY ( Paris )
CITY ( Rome )
CITY ( Athens )
CITY ( Oslo )
CITY ( Stockholm )
CITY ( Madrid )
CITY ( Amsterdam )

S ( s, sn, st, c )     =* CITY ( c )
```

P (*p*, *pn*, *pc*, *pw*, *c*) =* CITY (*c*)
J (*j*, *jn*, *c*) =* CITY (*c*)

b. Can't be done without appropriate scalar operators.

c. P (*p*, *pn*, Red, *pw*, *pc*) =* *pw* < 50

d. Can't be done without negation or aggregate operators.

e. S (*s1*, *sn1*, *st1*, Athens) AND
S (*s2*, *sn2*, *st2*, Athens) =* *s1* = *s2*

f. Can't be done without grouping and aggregation.

g. Can't be done without grouping and aggregation.

h. J (*j*, *jn*, *c*) =* S (*s*, *sn*, *st*, *c*)

i. J (*j*, *jn*, *c*) =* SPJ (*s*, *p*, *j*, *q*) AND S (*s*, *sn*, *st*, *c*)

j. P (*p1*, *pn1*, *pl1*, *pw1*, *pc1*) =* P (*p2*, *pn2*, Red, *pw2*, *pc2*)

k. Can't be done without aggregate operators.

l. S (*s*, *sn*, *st*, London) =* SP (*s*, P2, *q*)

m. P (*p1*, *pn1*, *pl1*, *pw1*, *pc1*) =*
P (*p2*, *pn2*, Red, *pw2*, *pc2*) AND *pw2* < 50

n.-o. Can't be done without aggregate operators.

p.-q. Can't be done (these are transition constraints).

24.8 *No answer provided.*

***** End of Chapter 24 *****

P A R T V I

O B J E C T S , R E L A T I O N S , A N D X M L

The introduction to Part VI in the book itself is more or less self-explanatory:

(Begin quote)

Like Chapter 20, the chapters in this part of the book rely heavily on material first discussed in Chapter 5. If you originally gave that chapter a "once over lightly" reading, therefore, you might want to go back and revisit it now (if you haven't already done so) before studying these chapters in any depth.

Object technology is an important discipline in the field of software engineering in general. It's therefore natural to ask whether it might be relevant to the field of database management in particular, and if so what that relevance might be. While there's less agreement on these questions than there might be, some kind of consensus does seem to be emerging. When object database systems first appeared, some industry figures claimed they would take over the world, replacing relational systems entirely; other authorities felt they were suited only to certain very specific problems and would never capture more than a tiny fraction of the overall market. While this debate was raging, systems supporting a "third way" began to appear: systems, that is, that combined object and relational technologies in an attempt to get the best of both worlds. And it now looks as if those "other authorities" were right: Pure object systems might have a role to play, but it's a niche role, and relational systems will continue to dominate the market for the foreseeable future——not least because those "object/relational" systems are really just relational systems after all, as we'll see.

More recently, one particular kind of object that's attracted a great deal of attention is *XML documents;* the problem of keeping such documents in a database and querying and updating them has rapidly become a problem of serious pragmatic significance. "XML databases"——that is, databases that contain XML documents and nothing else——are possible; however, it would clearly be preferable, if possible, to integrate XML documents with other kinds of data in either an object or a relational (or "object/relational") database.

Copyright (c) 2003 C. J. Date

The chapters in this part of the book examine such matters in depth. Chapter 25 considers pure object systems; Chapter 26 addresses object/relational systems; and Chapter 27 discusses XML.

(End quote)

Note: The book deliberately doesn't use the abbreviation "OO" very much. It also prefers "object" over "object-oriented" in adjectival positions.

***** End of Introduction to Part VI *****

Chapter 25

O b j e c t D a t a b a s e s

Principal Sections

- Objects, classes, methods, and messages
- A closer look
- A cradle-to-grave example
- Miscellaneous issues
- Summary

General Remarks

No "SQL Facilities" section in this chapter——discussion of the impact of "objects" on SQL is deferred to Chapter 26, *q.v.* There are, however, a few references to SQL in passing.

I have strong opinions on the subject of object databases, opinions that not everyone agrees with (and for that reason some instructors might find themselves out of sympathy with this chapter). Those opinions——stated in so many words at the end of Section 25.6——can be summed up as follows:

> The one good idea of objects is **proper data type support;** everything else, including in particular the notion of *user-defined operators,* follows from that basic idea.

(What's more, that idea is hardly new, but this point is unimportant.) *Note:* The foregoing should not be taken to mean that I think object databases have no role to play; rather, it means I think we need to be very clear on just what that role is. See the further discussion of this point in the notes on Section 25.6.

Be that as it may, the chapter is meant, first, as a tutorial on object concepts (as those concepts apply to database technology specifically); second, as a lead-in to the discussion of object/relational databases in Chapter 26. It shouldn't be skipped, though it might perhaps be condensed somewhat. Section 25.5 could be skipped.

Please note the following (paraphrased from reference [3.3]):

(Begin quote)

Copyright (c) 2003 C. J. Date

The label "object-oriented" (or just "object") is applied to a wide variety of distinct disciplines. It's used among other things to describe a certain *graphic interface* style; a certain *programming* style; certain *programming languages* (not the same thing as programming style, of course); certain *analysis* and *design* techniques; and, of course, a certain approach to *database management*. And it's quite clear that the term doesn't mean the same thing in all of these different contexts ... In this chapter, we're naturally interested in the applicability of object concepts and technology to database management specifically. *Please understand, therefore, that all remarks made in this chapter concerning object concepts and technology must be understood in this light;* we offer no opinion whatsoever regarding the suitability of object ideas in any context other than that of database management specifically.

(End quote)

Note too that the chapter describes object concepts——object *database* concepts, that is——**from a database perspective**. Much of the object database literature, by contrast, presents the ideas very much from a *programming* perspective instead; thus, it often simply ignores issues that the database community regards as crucial——*ad hoc* query, views, declarative integrity, concurrency, security, etc., etc. Part of the problem is that there aren't just two distinct *technologies* out there, there are two distinct *communities* as well. And the database community and the object community don't seem to understand each other, or each other's issues, very well. In particular, the object community doesn't seem to understand the database community's insistence on separating logical and physical, and it doesn't seem to understand the database community's emphasis on declarative solutions——for "business rules" in particular. And, to be very specific, it doesn't seem to understand the relational model (at least, such is my own personal experience).

Note the motivating discussions in Section 25.1, especially the rectangles example (forward pointer to Section 26.1). *Note:* The text says: "Convince yourself that [the original long SQL query] is correct." *No answer provided!*

25.2 Objects, Classes, Methods, and Messages

The table of rough equivalences in Fig. 25.3 (reproduced below) summarizes this section:

Object term	*Traditional term*

immutable object	value
mutable object	variable
object class	type
method	operator
message	operator invocation

Note carefully the discussion of **encapsulation** and some of the confusion that surrounds this term. Myself, I greatly prefer the term **scalar** (the two terms do mean the same thing, but *scalar* has a longer and more respectable pedigree).

Explain **public *vs.* private instance variables** very carefully (many people seem to be confused over this issue). Pure systems don't support public instance variables, but most systems aren't pure.

Mention **OIDs** but don't get into detail (yet).

25.3 A Closer Look

Explain **containment hierarchies** ("objects contain objects"——though, more usually, they contain *OIDs of* objects, not objects *per se*). *Note:* One reason (mentioned only briefly, later in the chapter) for choosing a containment hierarchy design is *performance*. An example of mixing logical and physical considerations?

Objects are really **tuples** (though probably tuples with RVAs, or something somewhat analogous to RVAs).

Object systems support a variety of **"collection" type generators** (LIST, BAG, etc.); another example of mixing logical and physical?

Discuss **object IDs** *vs.* "user keys" (but don't confuse OIDs and *surrogates*).

Discuss **class *vs.* instance *vs.* collection** and *"constructor functions." Caveat:* "Constructor functions" are *not* the same thing as *selectors*. See the notes on Section 25.6.

Note the cumbersome circumlocutions in this section——e.g.:

> "The effect of the ADD method invocation is to add the OID of the EMP object whose OID is given in the program variable E to the (previously empty) set of OIDs whose OID is given in the EMP_COLL object whose OID is given in the program variable ALL_EMPS."

In practice, of course, we don't really talk like this; we say, rather, things like "The ADD method adds employee E to the set of all employees." But this latter abbreviated form ***skips several levels of indirection***. It's OK to use such abbreviations if everyone understands what's really going on, but while people are learning I think it's better to spell it all out (tedious though it might seem to do so).

The parallel to PL/I (or any other language that supports "explicit dynamic variables") is illuminating if the audience has the appropriate background, but can be skipped otherwise.

Mention **class hierarchies** (unless Chapter 20 was skipped; either way, ***don't*** try to explain inheritance in depth at this juncture!).

25.4 A Cradle-to-Grave Example

Most books and papers on object databases show only snippets of code (or pseudocode), not whole programs. (Reference [25.35] is an exception.) But without looking at whole programs, or something close to whole programs, it's hard to get the big picture. The present section——which is, it might as well be admitted right away, more than a little tedious*——is intended to help in this regard. The details are messy but the section as a whole should be essentially self-explanatory.

———————

* That's part of the point, of course.

———————

The Smalltalk exanples could be replaced by equivalent examples in Java or C++ or whatever, if desired (though Java and C++ aren't as "pure" as Smalltalk, which is why the book uses Smalltalk in the first place).

The section doesn't discuss the point, but SET is an example of a ***union type*** (class) in the sense of Chapter 20. There are some mysteries involved in defining ESET, CSET, and the rest as subclasses of SET, but they aren't mentioned in the book and I wouldn't mention them in a live class, either.

The section closes by saying: "Note finally that REMOVE can be used to emulate a relational DROP operation——e.g., to drop the ENROLLMENT class. The details are left as an exercise." This

exercise is suitable for class discussion. Note that it will probably lead to a discussion of the *catalog*. See also Exercise 25.9 and Section 25.5. *No further answer provided*.

25.5 Miscellaneous Issues

This section could be skipped or condensed.

Originally, object systems couldn't do *ad hoc* query etc. (nor did they need to). Present-day systems can, but they do it via public instance variables——i.e., by "violating encapsulation," thereby undermining the whole point of objects! (See reference [25.31].) Note our own recommended approach to this issue (explained in the text). Note too the **important** rhetorical question: *What class is the query result?* If you don't have a good answer to this question, you don't really have a system (see Section 26.2 in the next chapter).

By the way: There's no objection to supporting "path expressions" that are merely shorthand for certain relational expressions. Rather, the objection is to being limited to using "path expressions" *only*——i.e., to being limited to traversing only predefined paths in the database (it's germane to observe that we used to be limited in exactly this way in IMS and other prerelational systems, and we know what problems that limitation led to).

Regarding *integrity:* The (procedural) object approach to this issue is a **giant** step backward!

Regarding *relationships:* In addition to the issues raised in the text, note the point (made previously in Chapter 14) that it's not a good idea to make a *formal* distinction between "objects" (= entities?) and relationships.

Regarding *database programming languages:* Some people, myself included, do like this idea, but of course it doesn't really have anything to do with objects. Indeed, **Tutorial D** is a database programming language——it makes no artificial and unnecessary distinctions between primary and secondary memory. Mention the business of *impedance mismatch* (though this term has several interpretations, none of them very precise).

Regarding *performance:* Self-explanatory. But note that (a) there's no reason why the techniques discussed——assuming they're a good idea——shouldn't be used in (e.g.) relational systems as well as object systems; (b) it could be argued that object systems achieve improved performance——to the extent they do——by "moving users closer to the metal."

Copyright (c) 2003 C. J. Date

"Is an object DBMS really a DBMS?" Self-explanatory. But the point, perhaps, is this: "Object DBMSs" do surely have a role to play; there are surely problems out there for which an "object DBMS" is the right solution. No argument here. No: The argument, rather, is simply that those "DBMSs" are ***not***——for all kinds of reasons——DBMSs in the sense in which the database community understands and uses that term. It might have been better not to call them DBMSs.

Reject the jingle *"persistence orthogonal to type"!*

25.6 Summary

For this chapter, alone out of the whole book, it seems worth including most of the summary section in these notes, because it really serves not just as a summary *per se* but also as a critical analysis of the material discussed and as a lead-in to what might constitute a "good" object model. So here goes (the following is reworded just a little from the original):

(Begin quote)

- **Object classes** (i.e., *types*): Obviously essential (indeed, they're the most fundamental construct of all).

- **Objects:** Objects themselves, both "mutable" and "immutable," are clearly essential——though I'd prefer to call them simply *variables* and *values,* respectively.*

———————

* Actually it might be argued that "mutable objects" aren't quite the same thing as variables in the classical sense. The one operator that *must* be available for a variable *V* is "assignment to *V*"——it's precisely the availability of that operator that makes *V variable!* But objects aren't required to have an associated assignment "method" (and indeed they typically don't); instead, such a method exists only if the class definer defines it.

———————

- **Object IDs:** Unnecessary, and in fact undesirable (at the model level, that is), because they're basically just pointers. Note too the argument, elaborated in the next chapter, that OIDs are fundamentally incompatible with a good model of inheritance. One problem——not the only one——is that

OIDs lead to the possibility of *shared variables,* a possibility that doesn't exist (nor do we want it to) in the relational world.

Note: Two points arise here:

1. Since I first wrote that sentence about shared variables (in the Instructor's Manual for the seventh edition), the possibility in question has been introduced into the *SQL* world. I regard this state of affairs as further evidence that the relational world and the SQL world are not the same ... Worlds apart, in fact.

2. *Don't* fall into the trap of thinking that if two distinct tuples in a relational database contain the same foreign key value and thus reference the same target tuple, that target tuple is a "shared variable." It isn't. It isn't a variable at all, in fact (tuples are *values*). See further discussion in the next chapter.

- **Encapsulation:** As explained in Section 25.2, "encapsulated" just means *scalar,* and I would prefer to use that term (always remembering that some "objects" aren't scalar anyway).

- **Instance variables:** First, *private* instance variables are by definition merely implementation matters and hence not relevant to the definition of an abstract model, which is what we're concerned with here. Second, *public* instance variables don't exist in a pure object system and are thus also not relevant. I conclude that instance variables can be ignored; "objects" should be manipulable solely by "methods" (see below).

- **Containment hierarchy:** We saw in Section 25.3 that containment hierarchies are misleading and in fact a misnomer, since they typically contain OIDs, not "objects." *Note:* A (nonencapsulated) hierarchy that really did include objects *per se* would be permissible, however, though usually contraindicated; it would be analogous, somewhat, to a relvar with relation-valued attributes (see Parts II and III of this book). Though we'd have to be careful yet again over the values *vs.* variables distinction ...

- **Methods:** The concept is essential, of course, though I would prefer to use the more conventional term *operators.*[*] Bundling methods with classes is *not* essential, however, and leads to several problems [3.3]; I would prefer to define "classes" (types) and "methods" (operators) separately, as in Chapter 5, and thereby avoid the notion of "target objects" and "selfish methods." (It's worth noting, incidentally, that the problems

introduced by bundling are not just syntactic ones. Again, see reference [3.3].)

---

* Another reason for avoiding the term "method" is that the term is used in the literature in two different senses: Sometimes it seems to mean the operator as seen by the user, sometimes it seems to mean the code that implements that operator. Yet another example of confusing model and implementation?

There are certain operators I'd insist on, too: Selectors (which among other things effectively provide a way of writing literal values of the relevant type), THE_ operators, assignment and equality comparison operators, and type testing and TREAT DOWN operators (see Chapter 20). I reject "constructor functions," however. Constructors construct *variables;* since the only kind of variable we want in the database is, specifically, the relvar, the only "constructor" we need is an operator that creates a relvar (e.g., CREATE TABLE, in SQL terms). Selectors, by contrast, select *values*. Also, of course, constructors return *pointers to* the constructed variables, while selectors return the selected values *per se*.

I would also stress the distinction between read-only and update operators (see Chapter 5).

- **Messages:** Again, the concept is essential, though I'd prefer to use the more conventional term *invocation* (and, again, I'd avoid the notion that such invocations have to be directed at some "target object" but instead treat all arguments equally).

- **Class hierarchy** (and related notions——inheritance, substitutability, inclusion polymorphism, and so on): Desirable but orthogonal (I see class hierarchy support, if provided, as just part of support for classes——i.e., types——*per se*).

- **Class *vs*. instance *vs*. collection:** The distinctions are essential, of course, but orthogonal (the *concepts* are distinct, and that's really all that needs to be said).

- **Relationships:** To repeat a point made earlier in these notes, it's not a good idea to treat "relationships" as a formally distinct construct——especially if it's only binary

relationships that receive such special treatment. I also don't think it's a good idea to treat the associated referential integrity constraints in some manner that's divorced from the treatment, if any, of integrity constraints in general (see below).

- **Integrated database programming language:** Nice to have, but orthogonal. However, the languages actually supported in today's object systems are typically ***procedural*** (3GLs) and therefore——I would argue——**nasty** to have (another giant step backward, in fact).

And here's a list of features that "the object model" typically ***doesn't*** support, or doesn't support well:

- ***Ad hoc* queries:** Early object systems typically didn't support ***ad hoc*** queries at all. More recent systems do, but they do so, typically, either by breaking encapsulation or by imposing limits on the queries that can be asked* (meaning in this latter case that the queries aren't really ***ad hoc*** after all).

_____

* I.e., by restricting them, via path expressions, to predefined paths in the database——as in IMS.

_____

- **Views:** Typically not supported (for essentially the same reasons that ***ad hoc*** queries are typically not supported). ***Note:*** Some object systems do support "derived" or "virtual" instance variables (necessarily public ones); e.g., the instance variable AGE might be derived by subtracting the value of the instance variable BIRTHDATE from the current date. However, such a capability falls far short of a full view mechanism——and in any case I've already rejected the notion of public instance variables.

- **Declarative integrity constraints:** Typically not supported (for essentially the same reasons that ***ad hoc*** queries and views are typically not supported). In fact, they're typically not supported even by systems that do support ***ad hoc*** queries.

- **Foreign keys:** The "object model" has several different mechanisms for dealing with referential integrity, none of which is quite the same as the relational model's more uniform

foreign key mechanism. Such matters as ON DELETE RESTRICT and ON DELETE CASCADE are typically left to procedural code (probably methods, possibly application code).

- **Closure:** What's (or, rather, *where's*) the object analog of the relational closure property?

- **Catalog:** Where's the catalog in an object system? What does it look like? Are there any standards? *Note:* These questions are rhetorical, of course. What actually happens is that a catalog has to be built by the professional staff whose job it is to tailor the object DBMS for whatever application it has been installed for, as discussed at the end of Section 25.5. (That catalog will then be *application-specific,* as will the overall tailored DBMS.)

To summarize, then, the good (essential, fundamental) features of the "object model"——i.e., the ones we really want to support——are as shown in the following table:

Feature	*Preferred term*	*Remarks*
object class	type	scalar & nonscalar; possibly user-defined
immutable object	value	scalar & nonscalar
mutable object	variable	scalar & nonscalar
method	operator	including selectors, THE_ ops, ":=", "=", & type test operators
message	operator invocation	no "target" operand

(End quote)

Answers to Exercises

25.1 We comment here on the term *object* itself (only; see the body of the chapter for the rest). Here are some "definitions" from the literature:

- "Objects are reusable modules of code that store data, information about relationships between data and applications, and processes that control data and relationships" (from a commercial product announcement; this sentence is hard enough to *parse,* let alone understand).

- "An object is a chunk of private memory with a public interface" (from reference [25.38]; the definition is true

enough, but hardly very precise; note too that it supports the position argued in reference [25.16] to the effect that the object model is really a ***storage*** model, not a data model).

- "An object is an abstract machine that defines a protocol through which users of the object may interact" (from the introduction to reference [25.42]).

- "An object is a software structure that contains data and programs" (from reference [25.24]; actually, objects ***don't*** contain programs, in general——***class-defining*** objects contain programs).

And my "favorite" (at the time of writing, at any rate) is this one:

- "***Object:*** A concrete manifestation of an abstraction; an entity with a well-defined boundary that encapsulates state and behavior; an instance of a class ... ***Instance:*** A concrete manifestation of an abstraction; an entity to which a set of operations can be applied and that has a state that stores the effects of the operations" (from reference [14.5]).*

Note that ***none*** of these "definitions" gets to what we would regard as the heart of the matter——***viz.,*** that an object is essentially just a ***value*** (if immutable) or a ***variable*** (otherwise).

_____

* If ***object*** and ***instance*** mean the same thing, why are there two terms? If they don't, what's the difference?

_____

It's worth commenting too on the notion that "everything's an object." Here are some examples of constructs that aren't objects (at least, they aren't in most object systems): ***instance variables; relationships*** (at least in ODMG [25.11]); ***methods; OIDs; program variables***. And in some systems (again including ODMG) ***values*** aren't objects either.

25.2 Some of the advantages of OIDs are as follows:

- They aren't "intelligent." See reference [14.10] for an explanation of why this state of affairs is desirable.

- They never change so long as the object they identify remains in existence.

- They're noncomposite. See references [14.11] and [19.8] for an explanation of why this state of affairs is desirable.

- Everything in the database is identified in the same uniform way (contrast the situation with relational databases).

- There's no need to repeat user keys in referencing objects. There's thus no need for any ON UPDATE rules.

Some of the ***dis***advantages——the fact that they don't avoid the need for user keys, the fact that they lead to a low-level pointer chasing style of programming, and the fact that they apply to "base" (nonderived) objects only——were discussed briefly in Sections 25.2-25.4. And the **huge** disadvantage, to the effect that they're incompatible with what I would regard as a "good" model of inheritance, is discussed in detail in the next chapter.

Possible OID implementation techniques include:

- Physical disk addresses (fast but poor data independence)

- Logical disk addresses (i.e., page and offset addresses; fairly fast, better data independence)

- Artificial IDs (e.g., timestamps, sequence numbers; need mapping to actual addresses)

25.3 See reference [25.15].

25.4 *No answer provided.*

25.5 We don't give a detailed answer to this exercise, but we do offer a few comments on the question of object database design in general. It's sometimes claimed that object systems make database design (as well as database use) easier, because they provide high-level modeling constructs and support those constructs directly in the system. (By contrast, relational systems involve an extra level of indirection: namely, the mapping process from real-world objects to relvars, attributes, foreign keys, and so on.) And this claim does have some merit. However, it overlooks the larger question: How is object database design done in the first place? The fact is, "the object model" as usually understood involves far more ***degrees of freedom***——in other words, more choices——than the relational model does; and I, at least, am not aware of any good guidelines that might help in making those choices. For example, how do we decide whether to represent, say, the set of all employees as an array, or a list, or a set (etc., etc.)? "A powerful data model needs a powerful design methodology

... and this is a ***liability*** of the object model" (paraphrased somewhat from reference [25.24]; I would argue that that qualifier "powerful" should really be "complicated").

25.6 ***No answer provided*** (it's straightforward, but tedious).

25.7 ***No answer provided*** (ditto).

25.8 ***No answer provided*** (ditto).

25.9 We don't give a detailed answer to this exercise, but we do make one remark concerning its difficulty. First, let's agree to use the term "delete" as a shorthand to mean "make a candidate for physical deletion" (i.e., by erasing all references to the object in question). Then in order to delete an object *X,* we must first find all objects *Y* that include a reference to *X;* for each such object *Y,* we must then either delete that object *Y,* or at least erase the reference in that object *Y* to the object *X* (by setting that reference to the special value (?) ***nil***). And part of the problem is that it isn't possible to tell from the data definition alone exactly which objects include a reference to *X,* nor even how many of them there are. Consider employees, for example, and the object class ESET. In principle, there could be any number of ESET instances, and any subset of those ESET instances could include a reference to some specific employee.

25.10 There are at least nine possible hierarchies:

```
S contains ( P contains ( J ) )
S contains ( J contains ( P ) )
S contains ( P and J )
P contains ( J contains ( S ) )
P contains ( S contains ( J ) )
P contains ( J and S )
J contains ( S contains ( P ) )
J contains ( P contains ( S ) )
J contains ( S and P )
```

"Which is best?" is unanswerable without additional information, but almost certainly all of them are bad. That is, whichever hierarchy is chosen, there'll always be numerous problems that are hard to solve in terms of that particular hierarchy.

25.11 First of all, there are the nine "obvious" designs discussed in the previous answer. But there are many other candidate designs as well——for example, an "SP" class that shows directly which suppliers supply which parts and also includes two embedded sets of projects, one for the supplier and one for the part. There's also a very simple design involving no (nontrivial)

hierarchies at all, consisting of an "SP" class, a "PJ" class, and a "JS" class.

25.12 The performance factors discussed were ***clustering, caching, pointer swizzling,*** and ***executing methods at the server***. All of these techniques are applicable to any system that provides a sufficient level of data independence; they are thus not truly "object-specific." In fact, the idea of using the logical database definition to decide what physical clustering to use, as some object systems do, could be seen as potentially ***undermining*** data independence. ***Note:*** It should be pointed out too that another very important performance factor, namely **optimization,** typically does ***not*** apply to object systems.

25.13 Declarative support, if feasible, is ***always*** better than procedural support (for everything, not just integrity constraints). In a nutshell, as pointed out several times earlier in this manual (and in the book), declarative support means the system does the work instead of the user. That's why relational systems support declarative queries, declarative view definitions, declarative integrity constraints, and so on.

25.14 See the discussion of relationships in Section 25.5.

***** End of Chapter 25 *****

Chapter 26

Object / Relational Databases

Principal Sections

- **The First Great Blunder**
- **The Second Great Blunder**
- Implementation issues
- Benefits of true *rapprochement*
- SQL facilities

General Remarks

At first blush, this chapter might be thought a little lightweight (at least, until we get to the section on SQL). But there's a reason for this state of affairs! The fact is, the label "object/relational" is, primarily, *vendor hype* ... As the text asserts:

> **A true "object/relational" system would be nothing more than a true *relational* system!**

For consider:

- "Object/relational," if it means anything at all, has to mean marrying (good) object ideas with relational ideas.

- We saw in Chapter 25 that "good object ideas" simply means *proper data type support*.

- The relational model *presupposes* proper data type support (that's what domains are, data types, as we saw in Chapter 5).

- So we don't have to do *anything* to the relational model——except implement it, an idea that doesn't seem to have been tried very much——in order to achieve the object functionality we desire.

It follows that much of the stuff one might have been led by vendor hype to expect in this chapter——the stuff regarding user-defined types and type inheritance in particular (or "data blades," or "data cartridges," etc.)——has already been discussed earlier in the book.

Copyright (c) 2003 C. J. Date

To repeat, a true "object/relational" system is really nothing more than a true *relational* system. But, of course, the meaning of the term "relational" has become polluted over the years, thanks to SQL, so a new label such as "object/relational" has become necessary, at least for marketing purposes.

Emphasize the point that the all too common misconception that relational systems can support only a limited number of very simple data types is exactly that——a misconception.

Note the "good" (relational) solution to the rectangles problem. (The book gives that solution in **Tutorial D;** producing an SQL analog is left as an exercise. *No answer provided*.)

The chapter should not be skipped.

26.2 The First Great Blunder

So are there any "true object/relational" systems? Well, the sad fact is that we can observe two *Great Blunders* being committed out there in the marketplace (and in research, too, I'm sorry to have to add). And any system that commits either of these blunders can hardly be said to be relational, or "object/relational." And just about every system available is committing the second blunder, if not the first as well ... Draw your own conclusions.

By the way, I recognize that *blunder* is a pretty strong term, but I'm not trying to win friends here; I think the mistakes are severe enough to merit the term. *Note added later:* As it says in the book itself, one reviewer of an early draft objected to the use of the term *blunder,* observing correctly that it isn't a term commonly found in textbooks. Well, I admit I chose it partly for its shock value. But if some system *X* is supposed to be an implementation of the relational model, and then——some 25 years after the relational model was first defined——somebody adds a "feature" to that system *X* that totally violates the prescriptions of that model, it seems quite reasonable to me to describe the introduction of that "feature" as a blunder.

The first blunder is described in the present section. It consists of *equating relvars and domains* (or tables and classes, if you prefer). I should immediately explain that, along with Hugh Darwen, I've been arguing against this false equation for several years, and it's probably true to say that few products are actually adhering to it any more (in other words, I'd like to feel our arguments didn't completely fall on deaf ears). As already noted, however, just about every product on the market seems to be committing the second blunder!——in fact, it's at least arguable

that the SQL standard commits it (see Section 26.6). In other words, (a) the first blunder seems to lead inevitably to the second (i.e., if you commit the first, you'll commit the second too), but (b) sadly, it's possible to commit the second even if you don't commit the first.

Explain the "crucial preliminary question" (and say why it's crucial). Work through the detailed example. Note carefully that the tables really contain **pointers to** tuples and relations, not tuples and relations as such. Note too that PERSON and EMP are "supertable" and "subtable," respectively, but not super***type*** and sub***type!***——in particular, there's no substitutability [3.3].

Showstopping criticisms of the equation "relvar = class":

- A relvar is a ***variable*** and a class is a ***type***. There's a huge logical difference here.

- A true object class has methods and no public instance variables (at least if it's "encapsulated"). By contrast, a relvar "object class" has public instance variables and only optionally has methods (it's definitely not "encapsulated"). So one has ***A*** and not ***B,*** while the other has ***B*** and only optionally has ***A!*** Another logical difference.

- There's yet another huge logical difference between the column definitions "SAL NUMERIC" and "WORKS_FOR COMPANY": NUMERIC is a data type, COMPANY is a relvar.

- People who advocate the equation "table = class" really mean "***base*** table = class." Another serious mistake (a violation of ***The Principle of Interchangeability,*** in fact).

Introducing pointers into relations (***The Second Great Blunder***——forward pointer to the next section) **undermines the conceptual integrity of the relational model**. "Conceptual integrity" is a useful idea, by the way, and it's worth spending a minute or two on it——with examples (see reference [3.3]). ***Note:*** There are plenty of ***bad*** examples in SQL! Here are a few:

- The interpretation of ***null*** depends on context:

  - Comparisons : value unknown
  - Outer join : value not applicable (?)
  - AVG () : value undefined
  - SUM () : value zero
  - Type BOOLEAN : third truth value

  etc., etc., etc.

- SQL tables are *bags* of rows, yet "bag union" etc. aren't directly supported, nor are they easily simulated

- SQL concepts aren't agreeably few, and many are downright *dis*agreeable: e.g., nulls, 3VL, left-to-right column ordering, duplicate rows, subtables and supertables, etc., etc.

Note very carefully the discussion of where *The First Great Blunder* might have come from! (A confidence trick?)

26.3 The Second Great Blunder

Don't mix pointers and relations! See reference [26.15] for detailed arguments in defense of this position (which really shouldn't *need* defending, but those who don't know history are doomed to repeat it ...).

Note the further analysis in this section of some of the ideas involved in the example in the previous section (most of which were rather confused, as it turns out). The first is, precisely (though implicitly), *mixing pointers and relations*. Key point: Pointers point to *variables,* not *values* (because variables have addresses* and values don't; recall that values "have no location in time or space"). Hence, if relvar *R1* includes an attribute whose values are pointers "into" relvar *R2,* then those pointers point to tuple *variables,* not to tuple *values*. **But there's no such thing as a tuple variable in the relational model**. (Relation *variables* contain relation *values,* and relation *values* can hardly be regarded as containing tuple *variables!* In fact, of course, as pointed out in the notes on Chapter 23 in connection with the special variable NOW, the notion of *any* kind of value containing *any* kind of variable is obviously nonsense, logically speaking.)

———————

* After all, a variable represents an abstraction of a chunk of storage.

———————

The quote from Ted Codd [6.2] is worth emphasizing. Also (to quote the text): "Actually there's another powerful argument against supporting pointers, one that Codd couldn't possibly have been aware of when he was writing reference [6.2]"——namely, *pointers and a good model of inheritance are incompatible*.* Go through the example (drawing pictures can help). *Note:* The example is expressed in **Tutorial D** style——not *really* **Tutorial D,**

because **Tutorial D** doesn't have any pointer support——but you might prefer to replace it by (e.g.) a Java equivalent.

———————

* To repeat a remark from the notes on Chapter 20, this fact implies that ***objects*** and a good model of inheritance are incompatible, since objects rely on pointers.

———————

Note the discussion of where the second blunder might have come from, too. The Wilkes quote is nice.

26.4 Implementation Issues

Mostly self-explanatory. Note the implication that, even though user-defined data type support might be thought of as simply an add-on to existing SQL support (and so it is, logically), it's certainly **not** just an add-on in implementation terms. That is, a good object/relational system can't be built by simply adding a new layer on top of an existing SQL implementation. Rather, the DBMS has to be ripped apart and rebuilt "from the ground up" (because good user-defined data type support affects so many different components of the system). These observations might help in the evaluation and comparison of commercial offerings in this arena.

26.5 Benefits of True *Rapprochement*

Stonebraker's "DBMS classification matrix" is, of course, very simplistic, but it can serve as a useful organizing principle for discussion. Note Stonebraker's position that "object/relational systems are in everyone's future"; they're not just a passing fad, soon to be replaced by some other briefly fashionable idea. And I agree with this position, strongly——though I'm not sure I agree with Stonebraker on exactly what an object/relational system ***is!*** In particular, Stonebraker never states explicitly that a true "object/relational" system would be nothing more than a true ***relational*** system, nor does he ever discuss "the equation wars" (domain = class ***vs.*** table = class).

In addition to the benefits listed, it would be a shame to walk away from nearly 35 years of solid relational R&D.

26.6 SQL Facilities

To quote: "SQL:1999's object/relational features are the most obvious and extensive difference between it and its predecessor SQL:1992." Remind students that:

- SQL supports two kinds of user-defined types, DISTINCT types and structured types, both of which can be used as a basis for defining columns in base tables (among other things)——see Chapter 5.

- Structured types (only) can also be used as the basis for defining "typed tables"——see Chapter 6.

- SQL-style inheritance applies to structured types (only)——see Chapter 20.

Now we need to add to the foregoing some discussion of (a) the ***REF type generator*** and (b) ***subtables and supertables***.

Regarding ***REF types:*** Explain carefully why "typed tables" aren't really "of" the type they're said to be! Note the "self-referencing column" terminology. ***Note:*** This stuff is very hard to explain, because it doesn't really make sense when you get right down to it. Note the footnote regarding circularity ... In the last analysis, it all boils down to a confusion over values ***vs.*** variables. Note the ambiguity (confusion?) over encapsulation, too.

Show some data manipulation examples. Explain (SQL-style) dereferencing. "Typed tables" have two different types at the same time! "It's all just shorthand, really" (?).

This section includes the following text: "Note the NOT NULL specifications on the columns of table EMP. Specifying that the columns of table DEPT also have nulls not allowed is not so easy! The details are left as an exercise." ***Answer:*** Explicit constraints will be necessary——e.g.:

```
CREATE ASSERTION BUDGET_NOT_NULL
       CHECK ( NOT EXISTS ( SELECT *
                            FROM   DEPT
                            WHERE  DEPT.BUDGET IS NULL ) ) ;
```

Regarding ***subtables and supertables:*** Explain the semantics and "behavior." What's this feature ***for?*** Good question! Note that the only things that might be useful at the ***model*** level can be achieved via views anyway [3.3]; in fact, we could ***implement*** subtables and supertables with views. It's my own strong suspicion that the real point is to allow a subtable and

supertable to be stored as a single table on the disk. If I'm right here, then it's a horrible model *vs.* implementation confusion.

"[If] SQL does not quite commit The Two Great Blunders, it certainly sails very close to the wind ...": Explain. Note (a) the "extent" stuff, (b) the fact that SQL suffers from the problem discussed earlier under the heading "Pointers and a Good Model of Inheritance Are Incompatible." What's the justification for all of this stuff? Note the following annotation to reference [26.21] (that reference consists of an overview of the additions made to the standard with SQL:1999):

(Begin quote)

[When] this article first appeared, Hugh Darwen and the present author wrote to the *SIGMOD Record* editor as follows: "With reference to [the subject article]——in particular, with reference to the sections entitled 'Objects ... Finally' and 'Using REF Types'——we have a question: What useful purpose is served by the features described in those sections? To be more specific, what useful functionality is provided that can't be obtained via features already found in SQL:1992?" Our letter wasn't published.

(End quote)

Answers to Exercises

26.1 See Section 26.1.

26.2 Essentially the same thing happens as happened with the code from Section 26.3 (whichever of the three possibilities that might have been); the overall conclusion is the same, too.

26.3 An analogous problem does *not* arise with foreign keys. In order to show why, we return to the original example from Section 26.3. *Note:* The following explanation is taken from reference [3.3], Appendix G, pages 421-422.

```
VAR E  ELLIPSE ;
VAR XC REF_TO_CIRCLE ;

E   :=  CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
XC  :=  TREAT_DOWN_AS_REF_TO_CIRCLE ( REF_TO ( E ) ) ;
THE_A ( E )  :=  LENGTH ( 6.0 ) ;
```

Ignoring irrelevancies, a relational analog of this example might look something like this:

```
VAR R1 ... RELATION { K ELLIPSE ... } KEY { K } ;

VAR R2 ... RELATION { K CIRCLE ... }
           FOREIGN KEY { K } REFERENCES R1 ;
```

For simplicity, assume no "referential actions"——cascade update, etc.——are specified (this simplifying assumption doesn't materially affect the argument in any way). Note that every K value in R1 that "matches" some K value in R2 must be of type CIRCLE, not just of type ELLIPSE.

Now let's insert a relation containing just one tuple into each of the two relvars:

```
INSERT R1
RELATION {
TUPLE { K CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) } } ;

INSERT R2
RELATION {
TUPLE { K CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) } } ;
```

Finally, let's try to update the tuple in R1:

```
UPDATE R1 { THE_A ( K ) := LENGTH ( 6.0 ) } ;
```

This UPDATE attempts to update the circle in the single tuple in R1 to make it of type ELLIPSE (we're speaking pretty loosely here, of course!). If that attempt were to succeed, the K value in R2 would refer to a "noncircular circle"——but that attempt does *not* succeed; instead, the UPDATE fails on a referential integrity violation.

Note: It's true that run-time errors can occur——referential integrity errors, to be precise——but run-time integrity violations are *always* possible, in general. At least we do have a system in which S by C and G by C are supported, type constraints are supported too, and noncircular circles and the like can't occur. (And run-time *type* errors specifically can occur only in the context of TREAT DOWN.)

26.4 Yes and no (probably more no than yes). *No further answer provided*.

26.5 It might make sense, but the variable won't be automatically maintained (i.e., if the row the variable points to is deleted, it'll be up to the user to realize that the variable now contains a dangling reference and deal with it appropriately).

26.6 *No answer provided* (it's tedious but essentially "straightforward").

26.7 *No answer provided*.

26.8 *No answer provided*.

***** End of Chapter 26 *****

Chapter 27

The World Wide Web and XML

Principal Sections

- The Web and the Internet
- An overview of XML
- XML data definition
- XML data manipulation
- XML and DBs
- SQL facilities

General Remarks

Nick Tindall of IBM was the original author of this chapter. You probably don't want to skip it.

There's a huge amount of interest these days in the Web, the Internet, and XML (trite but true observation). And there's a huge amount of material currently available on these topics, in all kinds of places. Comparatively little of that material seems to be written from a database perspective, however——and what little there is on database issues usually seems to come from people not knowledgeable in database technology. As a consequence, although XML in particular clearly does have implications for databases, the true nature of those implications doesn't seem to be well understood.* Indeed, there are some people who think XML is going to take over the database world completely——all databases will become XML databases, SQL will disappear (or be subsumed by XML), the relational model just won't be relevant any more, and on and on. Pretty strong claims for something that started out to be, in essence, nothing more than an approach to the **data interchange** problem! (To quote the XML specification [27.25], the original purpose of XML was "to allow generic SGML to be served, received, and processed on the Web like HTML.")

---

* In this connection, see the annotation to reference [27.3].

Copyright (c) 2003 C. J. Date

My own opinions regarding those "pretty stong claims" is summed up in the subsection "XML Databases" at the end of Section 27.6. To quote:

> "[We] saw in Chapter 3 that the relational model is both necessary and sufficient to represent any data whatsoever. We also know there's a huge investment in terms of research, development, and commercial products in what might be called *relational infrastructure* (i.e., support for recovery, concurrency, security, and optimization——not to mention integrity!——and all of the other topics we've been discussing in this book). In our opinion, therefore, it would be unwise to embark on the development of a totally new kind of database technology when there doesn't seem to be any overwhelming reason to do so ... Not to mention the fact that any such technology would obviously suffer from problems similar to those that hierarchic database technology already suffers from (see, e.g., Chapter 13 of reference [1.5] or the annotation to references [27.3] and [27.6])."

Note here the reference to hierarchic database technology, by the way. XML documents *are* hierarchic; XML databases (by which I mean what are sometimes called "native" XML databases) are thus hierarchic databases, and all of the old arguments against hierarchic databases apply directly (just as they do to object databases, as discussed in Chapter 25). In this connection, see the annotation to reference [27.6].

The purpose of this chapter, then, is to try to get at the true nature of what the relationship is or should be between XML and database technology. No prior knowledge of XML is assumed.

27.2 The Web and the Internet

You can skip this section if you like——most people are familiar with the Web and the Internet these days. The purpose of the section is simply to present some background for the XML discussions to follow, and in particular to introduce a few terms that are thrown around a lot in such contexts without (sometimes) a very clear understanding of what they mean: hypertext, URL, HTML, HTTP, website, web page, web browser, web server, web crawler, search engine, etc.

27.3 An Overview of XML

To repeat, no prior knowledge of XML is needed for this chapter. That's why there are three sections on XML *per se:* this overview section, plus the next two on XML data definition and XML data manipulation, respectively. Note that there's very little on databases as such in these three sections. However, they're definitely written from a database viewpoint: They downplay some aspects——e.g., namespaces, stylesheets——that XML aficionados might think are important but database people probably don't; at the same time, they emphasize others——e.g., integrity, data types——that XML people don't seem to be very interested in but database people are (or should be!). As a consequence, I think you should at least "hit the highlights" of these three sections, even if your audience is already "XML-aware." In the case of the present section, the highlights are as follows:

- An **XML document** is a document created using XML facilities (loose definition; the definition is loose because XML documents are really created using, not XML *per se,* but rather some "XML derivative"; XML is really a *meta*language or, more precisely, a *metameta*language).

- Explain **elements; tags** (note that there's some confusion over the precise meaning of this term); **attributes; empty elements**. *Note:* This latter is another misnomer, really——an empty element is an element that contains an empty character string (which isn't the same as being empty, which would mean it contains nothing at all), and it often has attributes too.

- Mention development history: proprietary——and somewhat procedural——markup languages such as Script; then GML; Standard GML; HTML; XML. XML has not exactly met its original goal of replacing HTML, but it *has* been widely used for other purposes. That's why there's a need to keep XML data in databases. The DRAWING example is worth discussing (note the message, implicit in that example, that an XML document might very reasonably appear in a relational database as an attribute value within some tuple).

- Definitely discuss the PartsRelation example. Point out that (to quote) "the XML document ... isn't a very faithful representation of a parts relation, because it imposes a top-to-bottom sequence on the tuples and a left-to-right sequence on the attributes of those tuples (actually lexical sequence in both cases)." By contrast, XML attributes are unordered, so it might be preferable to represent relational attributes by XML ditto. Note, however, that the "XML collection" support in SQL/XML (see Section 27.7) does map relational attributes to XML elements, not attributes; SQL/XML is thus subject to the foregoing criticism, and it isn't "stacking the deck" to introduce such an example.

- Explain "XML derivatives" (the official term is "XML applications") and **XML document structure** (nodes). The root or **document** node does *not* correspond to the document root element (trap for the unwary). Explain **the information set** ("infoset"); mention DOM. Another quote: "It might help to point out that the infoset for a given document is very close to being a *possrep* for that document, in the sense of Chapter 5."

- Introduce **"the semistructured data model"** (I set this phrase in quotes because I'm highly skeptical, or suspicious, regarding that term "semistructured"*). **Relations are no more and no less "structured" than XML documents are.** Anything that can be represented as an XML document can equally well be represented relationally——possibly as a tuple, possibly as a set of tuples, possibly otherwise. See Exercise 27.26.

_____

* I'm also highly skeptical, or suspicious, regarding the term "schemaless," which is also much encountered in this context. See Exercise 27.27.

_____

- Indeed, as the book says, I see no substantial difference between "the semistructured model" and the old-fashioned **hierarchic** model (or, at least, the structural aspects of the hierarchic model). See Exercise 27.29.

27.4 XML Data Definition

Regarding **DTDs,** explain:

- The fact that they're part of the XML standard *per se*.

- The revised PartsRelation example, with its DTD.

- **Well-formedness**. *Note:* This term is slightly strange, in a way, since if a document isn't well-formed then it just isn't an XML document in the first place (all XML documents are well-formed, by definition). It's kind of like saying a relation isn't well-formed if it involves (say) a left-to-right ordering to its attributes; if it involves a left-to-

right ordering to its attributes, then it just isn't a relation.

- **Validity** (= conformance to some DTD).

- DTD support for integrity constraints: legal values, attributes of type ID and IDREF.

- Limitations of DTDs (with respect to integrity in particular).

Regarding **XML Schema,** explain:

- XML schemas are XML documents.

- The further revised PartsRelation example, with its schema.

- **Types** and **type constraints** (but they're really just PICTUREs, á la COBOL, in traditional programming language terms).

- Mention additional advantages ***vis-á-vis*** DTDs.

- Mention **schema validation**.

Finally, a word on "metametalanguages": XML defines (among other things) the rules for constructing DTDs; and a DTD in turn is a metalanguage that defines the rules for constructing conforming documents. So a DTD is a metalanguage, and XML itself is, as claimed, really a metametalanguage. A quote: "[All] of those rules are, primarily, ***syntax*** rules; neither XML in general nor a given DTD in particular ascribes any meaning to documents created in accordance with those rules."

27.5 XML Data Manipulation

XQuery:

- Subsumes **XPath,** which we'll get to in a minute.

- Is read-only (= no updating——it really is just for query).

- Is large and complex——not to mention somewhat procedural, and (in my opinion) badly designed in certain respects ("from the folks who brought you SQL ...?").

- ***Doesn't operate on XML documents, as such, at all!*** This is the sort of thing that happens if you focus purely on data structure first (ignoring operators), and then try to graft

operators on afterward; in other words, if you're not a database person and you don't know about data models, or if you're not a languages person and you don't know about types. To elaborate: There was an attempt for a while to define an "XML document algebra" (retroactively), but the task was obviously impossible. To be specific, if X is an XML document, then X MINUS X would have to return something that isn't an XML document (there's no such thing as a completely empty XML document——there has to be a root element, even if that element itself is "empty"). So the algebra had to be defined, not over XML documents as such, but over certain ***abstractions*** of such documents, called ***sequences*** (and an empty sequence was legal). Some of the ideas of that algebra were subsequently incorporated into XQuery. ***Note:*** There are other reasons, noted in the chapter, why XQuery can't deal with XML documents as such, but the foregoing is a conceptually important one.

We need to cover XPath first. Explain **path expressions** (relate to path expressions in object systems; XML documents are like OO containment hierarchies!). "Manual navigation" look and feel. Currency ("context nodes"). A quote: "One problem with XPath is that it's fundamentally just an addressing mechanism; its path expressions can navigate to existing nodes in the hierarchy, but they can't construct nodes that don't already exist." Analogy with a "relational" language that supports restrictions and projections but not joins. Hence XQuery, which does have the ability to construct new nodes. Explain:

- Similarities and differences between XQuery expressions and ***relational calculus*** ditto.

- Similarities and differences between XQuery expressions and ***nested loops in a 3GL***. In my opinion, the parallels here are stronger. Note in particular that XQuery effectively hand-codes joins; note too that the particular nesting used in that hand-coding affects the result ("*A* JOIN *B*" and "*B* JOIN *A*" are logically different!).*

———————

* Part of the problem, it seems to me, is that ***sequences*** are the wrong abstraction; sets would have been better. Of course, this point is one large part of the old argument between hierarchies and relations. Once again, those who don't know history are doomed to repeat it?

———————

- **FLWOR expressions** in general (albeit in outline only). Difference between *for* and *let*. The fact that *order by* precedes *return* needs some explanation.

- At least one nontrivial hierarchic example.

A question: Is there any notion of *completeness* in XQuery, analogous to *relational* completeness in the relational world?

27.6 XML and DBs

Two requirements:

- Store XML data in databases and retrieve and update it.

- Convert "regular" (nonXML) data to XML form.

Regarding the first:

1. We might store the entire XML document as the value of some attribute within some tuple.

2. We might *shred* the document (technical term!) and represent various pieces of it as various attribute values within various tuples within various relations.

3. We might store the document not in a conventional database at all, but rather in a "native XML" database (i.e., one that contains XML documents as such instead of relations).

The third possibility has already been dismissed in these notes——though of course commercial products do exist that embrace that approach. The first possibility (documents as attribute values or "XML column") was touched on in the DRAWING example in Section 27.3; we haven't discussed the second possibility previously.

To elaborate on "XML column":

- Define a **new data type,** say XMLDOC, values of which are XML documents; then allow specific attributes of specific relvars to be of that type.

- Tuples containing XMLDOC values can be inserted and deleted using conventional **INSERTs** and **DELETEs**. XMLDOC values within such tuples can be replaced in their entirety using conventional **UPDATEs**. XMLDOC values can participate in **read-**

only operations in the conventional manner (SELECT and WHERE clauses, in SQL terms, loosely speaking).

- Type XMLDOC will have its own **operators** to support retrieval and update capabilities on XMLDOC-valued attributes at a more fine-grained level (e.g., at the level of individual elements or individual XML attributes). For retrieval, the operators might be like those of XQuery (they might even be invoked by means of an "escape" to XQuery).

"XML column" is appropriate for *document-centric* applications.

To elaborate on the second possibility——**shred and publish**, *aka* "XML collection":

- No new data types; instead, XML documents are "shredded" into pieces and those pieces are stored as values of various relational attributes in various places in the database.

- Hence, the DB doesn't contain XML documents as such. *The DBMS has no knowledge of such documents.* The fact that certain values in the database can be combined in certain ways to create such documents is understood by some application program (perhaps a web server), not by the DBMS.

- Since that application program can create an XML document from regular data, we've now met the second of our original objectives: We have a means of taking regular (nonXML) data and converting it to XML form (*publishing*): **XML views** of nonXML data (publishing for retrieval, shredding for update). Relate to *ANSI/SPARC architecture:* Hierarchic external level defined over relational conceptual level.

"XML collection" is appropriate for *data-centric* applications.

27.7 SQL Facilities

"SQL/XML" will probably be part of SQL:2003. It includes both "XML collection" and "XML column"——though just *why* it includes the first of these is very unclear to me, since (as we saw in the previous section) XML collection support has nothing to do with the DBMS, and SQL is supposed to a standard that relates to DBMSs (meaning functionality that DBMSs are supposed to support).

Briefly describe the XML collection support (XML views, retrieval only; equivalently, publishing only, no shredding). Discuss the simplified parts example. Several mysteries here! E.g., what about keys? What about user-defined types? What about NOT NULL specifications? More generally, what about integrity

constraints of any kind? Also, observe that (as noted earlier) publishing imposes an order on the tuples (rows in SQL).

Regarding the XML column support: Well, actually there isn't much. Mention **type XML,** plus operators to produce values of that type from conventional SQL data (e.g., **XMLGEN**). But almost no operators are defined for type XML——not even equality![*] "However, this state of affairs is likely to be corrected by the time SQL/XML is formally ratified."

_____

[*] In case anyone asks, note that XMLGEN is *not* an operator for type XML! It *returns* a value of type XML, but it operates on conventional SQL data.

_____

Sketch the **proprietary support** as outlined in the chapter, just to give an idea of the kind of functionality we might eventually expect to see in SQL/XML (as well as illustrating the kind of functionality already supported in some commercial products). See also Exercise 27.25.

Answers to Exercises

27.1 Some of the following definitions elaborate slightly on those given in the book *per se*.

- An **attribute** (in XML) is an expression of the form *name*="*value*"; it appears in a start tag or an empty-element tag, and it provides additional information for the relevant element.

- An **element** consists of a start tag, an end tag, and the "content" appearing between those tags. The content can be character data or other elements or a mixture of both. If the content is the empty string, the element is said to be empty, and the start and end tags can be combined into a single special tag, called an empty-element tag.

- **HTML** (Hypertext Markup Language) is a language for creating documents——in particular, documents stored on the Web——that include instructions on how they're to be displayed on a computer screen. HTML is an SGML derivative (i.e., it's defined using the facilities of SGML).

- **HTTP** (Hypertext Transfer Protocol) is a protocol for transmitting information over the Web. It's based on a request-response pattern: A client program establishes a connection with a server and sends a request to the server in a standard form; the server then responds with status information, again in a standard form, and optionally the requested information.

- The **Internet** is a supernetwork (actually a network of networks) of interconnected computers, communicating with each other via a common transmission and communication protocol called TCP/IP. Users have a variety of tools available for locating information and sending and receiving it over the Internet.

- **Markup** is metadata included in a document that describes the document content and optionally specifies how that content should be processed or displayed. Markup is typically distinguished from document content by "trigger" characters that indicate the start and end of pieces of markup——for example, semicolons or (as in XML) angle brackets.

- A **search engine** is a program that searches the Web for data that includes certain specified search arguments.

- **SGML** (Standard GML) is a standard form of GML (Generalized Markup Language). SGML and GML are metalanguages for defining specific markup languages. For example, HTML is a markup language defined using SGML (i.e., it's an SGML *derivative*).

- A **tag** is a piece of markup providing information about, and usually introducing or terminating, some fragment of textual information in a document. XML in particular defines three kinds of tags: start tags, end tags, and the special empty-element tag.

- A **URL** (Uniform Resource Locator) is the identifier of some resource available via the Internet. URLs have the general form:

  <scheme>:<scheme-specific part>

  The *<scheme>* identifies the relevant "scheme" or protocol in use (e.g., *http*); it determines how the *<scheme-specific part>* is to be interpreted.

- A **web browser** is a program that allows information to be retrieved from or submitted to the Web. Retrieved information

is displayed as web pages in graphical windows on the display screen.

- A **web crawler** is a continuously running program that analyzes and indexes web pages, with a view to speeding up subsequent searches for the information those pages contain.

- A **web page** is a unit of information, typically expressed in HTML, either stored on the Web or (possibly) manufactured on demand.

- A **web server** consists of a specialized computer and associated software whose role is to provide web content, particularly web pages, upon receiving requests from Web users. *Note:* The term is also used (and indeed was used in the body of the chapter) to refer to the software component alone.

- A **website** consists of a collection of related web pages, one of which (the *home* page) allows the user to navigate to the others.

- The **World Wide Web** is the agggregate of information stored on the Internet, together with the associated Web standards for interfaces and protocols by which that information can be stored, processed, and transmitted.

- **XML** is a proper subset of SGML. Its purpose is "to allow generic SGML to be served, received, and processed on the Web like HTML" (reworded slightly from reference [27.25]). It's really a metametalanguage (see Section 27.4); that is, it's a language for defining languages for defining languages (these last being markup languages specifically).

- An **XML derivative** (or "XML application") is a specific markup language, such as the Wireless Markup Language (WML) or Scalar Vector Graphics (SVG), that's defined using XML.

- **XML Schema** is an XML derivative whose purpose is to support the definition (i.e., of structure and content) of documents constructing using other XML derivatives.

- **XPath** is a language for addressing parts of an XML document. XPath is designed to be embedded as a sublanguage inside "host" languages such as XQuery and XSLT. XPath also has a natural subset, consisting of *path expressions,* that can be used by itself for a limited form of pattern matching——i.e., testing whether a given node matches a given pattern.

- **XQuery** is a query language, somewhat procedural in nature, for XML documents (more precisely, for a certain abstract form of such documents). An XQuery expression can access any number of existing documents; it can also construct new ones. At the time of writing, however, it provides no update facilities.

27.2 XML is a proper subset of SGML. The purpose of both is, loosely, to support the definition of other languages. HTML is a language whose definition is expressed in SGML; thus, SGML is the metalanguage for HTML. Similarly, XML is the metalanguage for languages such as Scalar Vector Graphics (SVG) that are defined using XML.

However, XML and SGML also include the specification of a document type definition (DTD) language, whose purpose is to specify some of the rules for languages defined using XML and SGML. So XML and SGML define a language for defining other languages, and they're thus really metametalanguages. In fact, starting with either XML or SGML, it's possible to construct an arbitrarily deep hierarchy of languages and metalanguages.

27.3 The following answer has been simplified in a variety of ways in the interest of brevity; for example, chapter and section numbers have been omitted, as have page numbers.* But what's left should be adequate to give the general idea.

---

* Because elements appear in a specific order, however, chapter and section numbers, at least, can be ***derived*** from the XML representation. Page numbers, by contrast, obviously can't be.

```
<?xml version="1.0"?>
<!-- XML document representing the table of contents. -->
<!DOCTYPE Contents [
   <!ELEMENT Contents (Preface?, Part+, Appendixes*, Index)>
   <!ELEMENT Preface (#PCDATA)>
   <!ELEMENT Part (Chapter+)>
      <!ATTLIST Part title CDATA #REQUIRED>
   <!ELEMENT Chapter (Introduction, Section+, Summary,
                      Exercises?, Refs-Bib, Answers?)>
      <!ATTLIST Chapter title CDATA #REQUIRED>
   <!ELEMENT Introduction EMPTY>
   <!ELEMENT Section (#PCDATA)>
   <!ELEMENT Summary EMPTY>
```

```
      <!ELEMENT Exercises EMPTY>
      <!ELEMENT Refs-Bib EMPTY>
      <!ELEMENT Answers EMPTY>
      <!ELEMENT Appendixes (Appendix+)>
      <!ELEMENT Appendix (Introduction?, Section*)>
         <!ATTLIST Appendix title CDATA #REQUIRED>
      <!ELEMENT Index EMPTY>
   ]>
   <Contents>
      <Preface>Eighth Edition</Preface>
      <Part title="Preliminaries">
         <Chapter title="An Overview of Database Management">
            <Introduction/>
            <Section>What is a database system?</Section>
             <Section>What is a database?</Section>
            <Section>Why database?</Section>
            <Section>Data independence</Section>
            <Section>Relational systems and others</Section>
            <Summary/>
            <Exercises/>
            <Refs-Bib/>
         </Chapter>
         <Chapter title="Database System Architecture">
            ...
         </Chapter>
      </Part>
      <Part title="The Relational Model">
         ...
      </Part>
      <Appendixes>
            ...
         <Appendix title="SQL Expressions">
            <Introduction>
               ...
         </Appendix>
      </Appendixes>
      <Index/>
   </Contents>
```

Here's another possible answer. This one has fewer structural constraints and makes less use of features not covered in the body of the chapter.

```
   <?xml version="1.0"?>
   <!-- XML document representing the table of contents. -->
   <!DOCTYPE Contents [
      <!ELEMENT Contents (Preface?, Part+, Appendixes*, Index)>
      <!ELEMENT Preface (#PCDATA)>
      <!ELEMENT Part (Chapter+)>
         <!ATTLIST Part title CDATA #REQUIRED>
      <!ELEMENT Chapter (Section+)>
```

```
      <!ATTLIST Chapter title CDATA #REQUIRED>
   <!ELEMENT Section (#PCDATA)>
   <!ELEMENT Appendixes (Appendix+)>
   <!ELEMENT Appendix (Section*)>
      <!ATTLIST Appendix title CDATA #REQUIRED>
   <!ELEMENT Index EMPTY>
]>
<Contents>
   <Preface>Eighth Edition</Preface>
   <Part title="Preliminaries">
      <Chapter title="An Overview of Database Management">
         <Section>Introduction</Section>
         <Section>What is a database system?</Section>
          <Section>What is a database?</Section>
         <Section>Why database?</Section>
         <Section>Data independence</Section>
         <Section>Relational systems and others</Section>
         <Section>Summary</Section>
         <Section>Exercises</Section>
         <Section>References and bibliography</Section>
         <Section>Answers to selected exercises</Section>
      </Chapter>
      <Chapter title="Database System Architecture">
         ...
      </Chapter>
   </Part>
   <Part title="The Relational Model">
      ...
   </Part>
   <Appendixes>
         ...
      <Appendix title="SQL Expressions">
         <Section>Introduction</Section>
            ...
      </Appendix>
   </Appendixes>
   <Index/>
</Contents>
```

27.4 Revise either of the answers given above for Exercise 27.3 as follows:

1. Move the text between

   ```
   <!DOCTYPE Contents [
   ```

   and

   ```
   ]>
   ```

   to a separate file called Contents.dtd (say).

2. Replace the text

    <!DOCTYPE Contents [

    by

    <!DOCTYPE Contents SYSTEM "Contents.dtd">

    and delete the text

    ]>

The advantage of an external DTD is that such a DTD can more easily be shared by distinct documents.

27.5 An XML document is **well-formed** if and only if all three of the following are true: It's syntactically correct according to the XML specification; it complies with all of the well-formedness rules in that specification; and all documents it refers to, directly or indirectly, are well-formed in turn. Strictly speaking, a piece of text isn't an XML document at all unless it's well-formed, so an "ill-formed XML document" is a contradiction in terms.

An XML document is **valid** if and only if all three of the following are true: It's well-formed (which it must be, otherwise it's not an XML document at all); it has a DTD or a schema; and it follows all the rules specified in that DTD or schema. ***Note:*** Validation with respect to a schema (as opposed to a DTD) is known as ***schema validation***. The term ***validation*** without that "schema" qualifier refers to validation with respect to a DTD.

27.6 An **empty element** is an element whose content is the empty string. For example:

```
<EmptyExample></EmptyExample>
```

Equivalently:

```
<EmptyExample/>
```

Note: Although an element can be "empty," its tag(s) can contain attributes and/or white space, as here:

```
<EmptyExample attr="val" another="more" andSoOn=""/>
```

27.7 Yes, they are. See Chapter 25 for a critical discussion of containment hierarchies in general.

Copyright (c) 2003 C. J. Date

27.8 It's true that data definitions in SQL are expressed using a special "data definition language" (CREATE TABLE, etc.). However, those definitions are represented in the database just like any other data——i.e., by means of tables (actually tables in the catalog). As we saw in Exercise 6.16, moreover, the operators of that data definition language are all, in the final analysis, shorthand for certain conventional SQL operators or operator combinations that could in principle be applied directly to those catalog tables. So no, an analogous criticism does *not* really apply to SQL. Similar remarks apply to the relational model.

27.9

```
<?xml version="1.0"?>
<!-- This is an XML version of the Projects relation -->
<ProjectsRelation>
   <ProjectTuple>
      <JNUM>J1</JNUM>
      <JNAME>Sorter</JNAME>
<CITY>Paris</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J2</JNUM>
      <JNAME>Display</JNAME>
<CITY>Rome</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J3</JNUM>
      <JNAME>OCR</JNAME>
<CITY>Athens</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J4</JNUM>
      <JNAME>Console</JNAME>
<CITY>Athens</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J5</JNUM>
      <JNAME>RAID</JNAME>
<CITY>London</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J6</JNUM>
      <JNAME>EDS</JNAME>
<CITY>Oslo</CITY>
   </ProjectTuple>
   <ProjectTuple>
      <JNUM>J7</JNUM>
      <JNAME>Tape</JNAME>
<CITY>London</CITY>
   </ProjectTuple>
```

```
    </ProjectsRelation>
```

There's no direct way to enforce the desired uniqueness constraint on JNUM.

27.10

```
    <?xml version="1.0"?>
    <!-- Another XML version of the Projects relation -->
    <!DOCTYPE ProjectsRelation [
       <!ELEMENT ProjectsRelation (ProjectTuple*)>
          <!ATTLIST ProjectsRelation
              JNUM ID #REQUIRED
       JNAME CDATA #REQUIRED
       CITY CDATA #REQUIRED>
    ]>
    <ProjectsRelation>
       <ProjectTuple JNUM="J1" JNAME="Sorter" CITY="Paris"/>
       <ProjectTuple JNUM="J2" JNAME="Display" CITY="Rome"/>
       <ProjectTuple JNUM="J3" JNAME="OCR" CITY="Athens"/>
       <ProjectTuple JNUM="J4" JNAME="Console" CITY="Athens"/>
       <ProjectTuple JNUM="J5" JNAME="RAID" CITY="London"/>
       <ProjectTuple JNUM="J6" JNAME="EDS" CITY="Oslo"/>
       <ProjectTuple JNUM="J7" JNAME="Tape" CITY="London"/>
    </ProjectsRelation>
```

Regarding uniqueness of JNUM values, see Section 27.4, subsection "Attributes of Type ID and IDREF." As for the relative advantages and disadvantages of using attributes, here are some relevant considerations:

- Elements can contain links to other resources (using XLink and XPointer), attributes can't.
- Elements are ordered, attributes aren't.
- Elements can appear any number of times (including zero), attributes can't.
- Attributes can specify defaults, elements can't.
- Attributes can provide some limited support for referential integrity, elements can't.
- Attributes don't work very well for composite values such as arrays.

27.11 See Section 27.4, subsection "Attributes of Type ID and IDREF."

Copyright (c) 2003 C. J. Date

27.12 Schemas can be formulated in a variety of different ways. One extreme is to make all elements global (i.e., immediate children of the xsd:schema element), cross-referencing them as necessary. This approach is particularly useful when type or element definitions are to be shared; it helps avoid redundant and potentially inconsistent definitions. The other extreme, illustrated by the answer below, is to make just the root element global, defining all child elements to be contained within that root element (at some level). Note the need to repeat the definition of the Section element (because chapters and appendixes both have sections). Since the Section element is quite simple, however (involving as it does just data of type xsd:string), the repetition isn't all that burdensome.

```
<?xml version="1.0"?>
<!-- Schema for second answer to Exercise 27.3 -->
<!DOCTYPE xsd:schema SYSTEM
          "http://www.w3.org/2001/XMLSchema.dtd">

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="Contents">
      <xsd:complexType>
         <xsd:sequence>

            <xsd:element name="Preface" type="xsd:string"
                                        minOccurs="0"/>
            <xsd:element name="Part" maxOccurs="unbounded">
               <xsd:complexType>
                  <xsd:sequence>

                     <xsd:element name="Chapter">
                        <xsd:complexType>
                           <xsd:sequence>

                              <xsd:element name="Section"
                                   type="xsd:string"
                                   maxOccurs="unbounded"/>

                           </xsd:sequence>
                           <xsd:attribute name="title"
                                          type="xsd:string"/>
                        </xsd:complexType>
                  </xsd:element>

                  </xsd:sequence>
                  <xsd:attribute name="title"
                                 type="xsd:string"/>
               </xsd:complexType>
            </xsd:element>
```

```
                <xsd:element name="Appendixes">
                   <xsd:complexType>
                      <xsd:sequence>

                         <xsd:element name="Appendix">
                            <xsd:complexType>
                               <xsd:sequence>

                             <xsd:element name="Section"
                                          type="xsd:string"
                                          maxOccurs="unbounded"/>

                               </xsd:sequence>
                               <xsd:attribute name="title"
                                    type="xsd:string"/>
                            </xsd:complexType>
                          </xsd:element>

                      </xsd:sequence>
                   </xsd:complexType>
                </xsd:element>

        <xsd:element name="Index" type="xsd:string"/>

             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>

    </xsd:schema>
```

27.13 Consider the XML schema shown for PartsRelation documents in Section 27.4, subsection "XML Schema." The only change required is in the definition of the "complex type" called PartTupleType. Replace the second line of that definition——the start tag "<xsd:sequence>"——by:

```
    <xsd:all>
```

Also, replace the corresponding end tag "<xsd:sequence/>" by:

```
    <xsd:all/>
```

The effect of these changes is precisely that elements directly contained within a PartTuple element can appear in any order, as desired.

27.14 A type as usually understood is a set of values (i.e., all possible values of the type in question), along with an associated set of operators that can be applied to values and variables of the type in question (see Chapters 5 and 20 for further

explanation). In XML Schema, by contrast, a type, though it does have a specified set of values, has almost no operators! (To be specific, it does have "=", and possibly "<", but no others.) Thus, although they have names like "string," "boolean," "decimal," etc., all of which have an obvious *intuitive* meaning, the corresponding XML "types" are certainly not string, boolean, decimal (etc.) types as usually understood. In fact, as noted in the body of the chapter, XML Schema "type definitions" are really closer to the PICTURE specifications found in languages like COBOL and PL/I; i.e., all they really do is define certain *character-string representations* for the "types" in question.

27.15 *Infoset* is a contraction of "information set." Every XML document has one. The infoset for a given document can be thought of as an abstract representation* of that document as a hierarchy of nodes or *information items* [27.26], each of which has a set of named *properties* (e.g., parent, children, "normalized value"). A given infoset can be *augmented* with additional properties (discovered, e.g., during schema validation); indeed, XPath and XQuery are defined in terms of such an augmented infoset, the *Post Schema Validation Infoset* (PSVI). For further discussion, see Section 27.3, subsection "XML Document Structure."

_____

* It might be thought of as a "possible representation" in the sense of Chapter 5.

_____

27.16 A path expression in XPath and XQuery is an expression that, when evaluated, navigates through some specific infoset to some specific node or sequence of nodes (i.e., it returns a value that is a sequence of *information items*——see the answer to Exercise 27.15). It consists of a sequence of *steps,* each of which generates a sequence of nodes and then optionally eliminates some of those nodes via *predicates*. Each step thus returns a sequence of nodes, which become the context for the next step if any. For further discussion, see Section 27.5, subsection "XPath."

27.17 A FLWOR expression is a fundamental XQuery building-block. It consists of one or more of the following clauses (in sequence as indicated):

- A *for* clause, which binds variables iteratively to sequences of items selected by expressions with optional predicates

- A *let* clause, which binds variables (without iteration) to entire sequences of items selected by expressions as in the *for* clause

- A *where* clause, which applies filtering criteria to the items specified by the *for* and/or *let* clauses

- An *order by* clause, which imposes a sequence on the results generated by the *return* clause (see next)

- A *return* clause, which generates the resulting sequence(s) of items

As the foregoing indicates, the crucial difference between the *for* and *let* clauses is that the *for* clause binds the items in the specified sequence to the specified variable one at a time——in other words, iteratively——whereas the *let* clause binds the specified sequence to the specified variable as a whole, without any iteration.

In many cases, predicates and *where* clauses are equivalent. Predicates might be more "natural" when they apply to the current context. *Where* clauses are perhaps more general (they can refer to arbitrary nodes, etc.). By way of example, here are two formulations of query 1.1.9.1 Q1 from the W3C *XML Query Use Cases* document (see reference [27.29]):

- Using predicates:

```
<bib>
   {
      for $b in document("http://www.bn.com/bib.xml")/bib/book
       [publisher = "Addison-Wesley"][@year > 1991]
      return
         <book year="{ $b/@year }">
            { $b/title }
         </book>
   }
</bib>
```

- Using a *where* clause:

```
<bib>
   {
      for $b in document("http://www.bn.com/bib.xml")/bib/book
 where $b/publisher = "Addison-Wesley" and $b/@year > 1991
      return
         <book year="{ $b/@year }">
            { $b/title }
```

```
            </book>
      }
   </bib>
```

- Result (for both queries):*

```
   <bib>
      <book year="1994">
         <title>TCP/IP Illustrated</title>
      </book>
      <book year="1992">
         <title>Advanced Unix Programming</title>
      </book>
   </bib>
```

---

* We've altered the "official" result very slightly here for formatting reasons.

27.18

```
   <Result>
      { document("PartsRelation.xml")//PartTuple[NOTE] }
   </Result>
```

27.19

```
   <Result>
      { for $p in document("PartsRelation.xml")//PartTuple
                                          [@COLOR = "Green"]
        return <GreenPart> {$p} </GreenPart>
      }
   </Result>
```

27.20 The result looks like this:

```
   <Parts>
     6
   </Parts>
```

27.21

```
   <Result>
   { for $sx in document("SuppliersOverShipments.xml")//Supplier
     where document("PartsRelation.xml")//PartTuple
```

```
                                [PNUM = $sx//PNUM][@COLOR = 'Blue']
      return
         <Supplier>
            { $sx/SNUM, $sx/SNAME, $sx/STATUS, $sx/CITY }
         </Supplier>
    }
    </Result>
```

27.22 Since the document doesn't have any immediate child elements of type Supplier, the ***return*** clause is never executed, and the result is the empty sequence. ***Note:*** If the query had been formulated slightly differently, as follows——

```
    <Result>
    { for $sx in document("SuppliersOverShipments.xml")/
                 Supplier[CITY = 'London']
      return
        <whatever>
           { $sx/SNUM, $sx/SNAME, $sx/STATUS, $sx/CITY }
        </whatever>
    }
    </Result>
```

——then the result would have looked like this:

```
    <Result>
    </Result>
```

27.23 There appears to be no difference. Here's an actual example (query 1.1.9.3 Q3 from the W3C ***XML Query Use Cases*** document——see reference [27.29]):

- Query:

```
    <results>
    {
       for $b in document("http://www.bn.com/bib.xml")/bib/book,
           $t in $b/title,
           $a in $b/author
       return
          <result>
             { $t }
             { $a }
          </result>
    }
    </results>
```

- Query (modified):

```
    <results>
```

```
{
   for $b in document("http://www.bn.com/bib.xml")/bib/book,
       $t in $b/title,
       $a in $b/author
   return
      <result>
         { $t, $a }
      </result>
}
</results>
```

- Result (for both queries):*

```
<results>
   <result>
      <title>TCP/IP Illustrated</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
   </result>
   <result>
      <title>Advanced Unix Programming</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
   </result>
   <result>
      <title>Data on the Web</title>
      <author>
         <last>Abiteboul</last>
         <first>Serge</first>
      </author>
   </result>
      ...
</results>
```

---

* Again we've altered the "official" result very slightly for formatting reasons.

27.24 See Section 27.6.

27.25 The following observations, at least, spring to mind immediately:

- Several of the functions perform what is essentially ***type conversion***. The expression XMLFILETOCLOB ('BoltDrawing.svg'), for example, might be more conventionally written something like this:

  ```
  CAST_AS_CLOB ( 'BoltDrawing.svg' )
  ```

  In other words, XMLDOC should be recognized as a fully fledged type (see Section 27.6, subsection "Documents as Attribute Values").

- Likewise, the expression XMLCONTENT (DRAWING, 'RetrievedBoltDrawing.svg') might more conventionally be written thus:

  ```
  DRAWING := CAST_AS_XMLDOC ( 'RetrievedBoltDrawing.svg' ) ;
  ```

  In fact, XMLCONTENT is an ***update operator*** (see Chapter 5), and the whole idea of being able to invoke it from inside a read-only operation (SELECT in SQL) is more than a little suspect [3.3].

- Consider the expression XMLFILETOCLOB ('BoltDrawing.svg') once again. The argument here is apparently of type character string. However, that character string is ***interpreted*** (in fact, it is **dereferenced**——see Chapter 26), which means that it can't be just any old character string. In fact, the XMLFILETOCLOB function is more than a little reminiscent of the EXECUTE IMMEDIATE operation of dynamic SQL (see Chapter 4).

- Remarks analogous to those in the previous paragraph apply also to arguments like

  ```
  '//PartTuple[PNUM = "P3"]/WEIGHT'
  ```

  (see the XMLEXTRACTREAL example).

27.26 The suggestion is correct, in the following sense. Consider any of the PartsRelation documents shown in the body of the chapter. Clearly it would be easy, albeit tedious, to show a tuple containing exactly the same information as that document——though it's true that the tuple in question would contain just one component, corresponding to the XML document in its entirety. That component in turn would contain a list or sequence of further components, corresponding to the first-level content of the XML document in their "document order"; those

components in turn would (in general) contain further components, and so on. Omitted elements can be represented by empty sequences. Note in particular that tuples in the relational model carry their attribute types with them, just as XML elements carry their tags with them——implying that (contrary to popular opinion!) tuples too, like XML documents, are self-describing, in a sense.

27.27 The claim that XML data is "schemaless" is absurd, of course; data that was "schemaless" would have no known structure, and it would be impossible to query it——except by playing games with SUBSTRING operations, if we stretch a point and think of such game-playing as "querying"——or to design a query language for it.* Rather, the point is that the schemas for XML data and (say) SQL data are expressed in different styles, styles that might seem distinct at a superficial level but aren't really so very different at a deep level.

_____

* In fact, it would be a BLOB——i.e., an arbitrarily long bit string, with no internal structure that the DBMS is aware of.

_____

27.28 In one sense we might say that an analogous remark does apply to relational data. Given that XML fundamentally supports just one data type, *viz.*, character strings, it's at least arguable that the options available for structuring such data (i.e., character-string data specifically) in a relational database are exactly the same as those available in XML. As a trivial example, an address might be represented by a single character string; or by separate strings for street, city, state, and zip; or in a variety of other ways.

In a much larger sense, however, an analogous remark does *not* apply. First, relational systems provide a variety of additional (and genuine) data types over and above character strings, as well as the ability for users to define their own types; they therefore don't force users to represent everything in character-string form, and indeed they provide very strong incentives not to. Second, there's a large body of design theory available for relational databases that militates against certain bad designs. Third, relational systems provide a wide array of operators, the effect of which is (in part) that there's no logical incentive for biasing designs in such a way as to favor some applications at the expense of others (contrast the situation in XML).

27.29 This writer is aware of no differences of substance——except that the hierarchic model is usually regarded as including certain operators and constraints, while it's not at all clear that the same is true of "the semistructured model."

27.30 *No answer provided.*

***** End of Chapter 27 *****

A P P E N D I X E S

The following text speaks for itself:

(Begin quote)

There are four appendixes. Appendix A is an introduction to a new implementation technology called *The TransRelational™ Model*. Appendix B gives further details, for reference purposes, of the syntax and semantics of SQL expressions. Appendix C contains a list of the more important abbreviations, acronyms, and symbols introduced in the body of the text. Finally, Appendix D (online) provides a tutorial survey of common storage structures and access methods.

(End quote)

***** End of Introduction to Appendixes *****

Copyright (c) 2003 C. J. Date

Appendix A

The TransRelational™ Model

Principal Sections

- Three levels of abstraction
- The basic idea
- Condensed columns
- Merged columns
- Implementing the relational operators

General Remarks

This is admittedly only an appendix, but if I was the instructor I would certainly cover it in class. "It's the best possible time to be alive, when almost everything you thought you knew is wrong" (from *Arcadia,* by Tom Stoppard). The appendix is about a radically new implementation technology, which (among other things) does mean that an awful lot of what we've taken for granted for years regarding DBMS implementation is now "wrong," or at least obsolete. For example:

- The data occupies a fraction of the space required for a conventional database today.

- The data is effectively stored in many different sort orders at the same time.

- Indexes and other conventional access paths are completely unnecessary.

- Optimization is much simpler than it is with conventional systems; often, there's just one obviously best way to implement any given relational operation. In particular, the need for cost-based optimizing is almost entirely eliminated.

- Join performance is linear!——meaning, in effect, that the time it takes to join twenty relations is only twice the time it takes to join ten (loosely speaking). It also means that joining twenty relations, if necessary, is feasible in the first place; in other words, the system is *scalable*.

Copyright (c) 2003 C.J.Date

- There's no need to compile database requests ahead of time for performance.

- Performance in general is orders of magnitude better than it is with a conventional system.

- Logical design can be done properly (in particular, there is never any need to "denormalize for performance").

- Physical database design can be completely automated.

- Database reorganization as conventionally understood is completely unnecessary.

- The system is much easier to administer, because far fewer human decisions are needed.

- There's no such thing as a "stored relvar" or "stored tuple" at the physical level at all!

In a nutshell, the TransRelational model allows us to build DBMSs that——at last!——truly deliver on the full promise of the relational model. Perhaps you can see why it's my honest opinion that "The TransRelational[tm] Model" is the biggest advance in the DB field since Ted Codd gave us the relational model, back in 1969.

Note: We're supposed to put that trademark symbol on the term TransRelational, at least the first time we use it, also in titles and the like. Also, you should be aware that various aspects of the TR model——e.g., the idea of storing the data "attribute-wise" rather than "tuple-wise"——do somewhat resemble various ideas that have been described elsewhere in the literature; however, nobody else (so far as I know) has described a scheme that's anything like as comprehensive as the TR model; what's more, there are many aspects of the TR model that (again so far as I know) aren't like anything else, anywhere.

The logarithms analogy from reference [A.1] is helpful: "As we all know, logarithms allow what would otherwise be complicated, tedious, and time-consuming numeric problems to be solved by transforming them into vastly simpler but (in a sense) equivalent problems and solving those simpler problems instead. Well, it's my claim that *TR technology does the same kind of thing for data management problems*." Give some examples.

Explain and justify the name: *The TransRelational[tm] Model* (which we abbreviate to "TR" in the book and in these notes). Credit to Steve Tarin, who invented it. Discuss *data independence*

and the conventional "direct image" style of implementation and the problems it causes.

Note the simplifying assumptions: The database is (a) read-only and (b) in main memory. Stress the fact that these assumptions are made purely for pedagogic reasons; TR can and does do well on updates and on disk.

A.2 Three Levels of Abstraction

Straightforward——but stress the fact that the files are abstractions (as indeed the TR tables are too). Be very careful to use the terminology appropriate to each level from this point forward. Show but do not yet explain in detail the Field Values Table and the (or, rather, a) Record Reconstruction Table for the file of Fig. A.3. ***Note:*** Each of those tables is derived from the file independently of the other. Point out that we're definitely not dealing with a direct-image style of implementation!

A.3 The Basic Idea

Explain "the crucial insight": Field Values in the Field Values Table, linkage information in the Record Reconstruction Table. By the way, I deliberately don't abbreviate these terms to FVT and RRT. Students have so much that's novel to learn here that I think such abbreviations get in the way (the names, by contrast, serve to remind students of the functionality). ***Note:*** Almost all of the terms in this appendix are taken from reference [A.1] and do ***not*** appear in reference [A.2]——which, to be frank, is quite difficult to understand, in part precisely because its terminology isn't very good (or even consistent).

Regarding the ***Field Values Table:*** Built at load time (so that's when the sorting is done). Explain intuitively obvious advantages for ORDER BY, value lookup, etc. The Field Values Table is the only TR table that contains user data as such. Isomorphic to the file.

Regarding the ***Record Reconstruction Table:*** Also isomorphic, but contains ***pointers*** (row numbers). Those row numbers identify rows in the Field Values Table or the Record Reconstruction Table or both, depending on the context. Explain the zigzag algorithm. Can enter the rings (zigzags) anywhere! Explain simple equality restriction queries (binary search). TR lets us do a sort/merge join without having to do the sort!——or, at least, without having to do the ***run-time*** sort (explain). Implications for the optimizer: Little or no access path selection. Don't need indexes. Physical database design is simplified (in fact, it

should become clear later that it can be *automated,* given the logical design). No need for performance tuning. A boon for the tired DBA.

Explain how the Record Reconstruction Table is built (or you could set this subsection as a reading assignment). Not unique; we can turn this fact to our advantage, but the details are beyond the scope of this appendix; suffice it to say that some Record Reconstruction Tables are "preferred." See reference [A.1] for further discussion.

A.4 Condensed Columns

An obvious improvement to the Field Values Table ... but one with far-reaching consequences. Note the implications for update in particular (we're pretending the database is read-only, but this point is worth highlighting in passing). The compression advantages are staggering!——but note that we're compressing at the level of field values, not of bit string encodings ... Don't have to pay the usual price of extra machine cycles to do the decompressing!

Explain row ranges.* Emphasize the point that these are *conceptual:* Various more efficient internal representations are possible. Histograms ... The TR representation is all about **permutations and histograms**. Immediately obvious implications for certain kinds of queries——e.g., "How many parts are there of each color?" Explain the revised record reconstruction process.

———————

* Row ranges look very much like intervals as in Chapter 23. But we'll see in the next section that we sometimes need to deal with *empty* row ranges, whereas intervals in Chapter 23 were always nonempty.

———————

A.5 Merged Columns

An extension of the condensed-columns idea (in a way). Go through the bill-of-materials example. Explain the implications for join! In effect, we can do a sort/merge join without doing the sort *and without doing the merge, either!* (The sort and merge are done at load time. Do the heavy lifting ahead of time! ... As with logarithms, in fact.)

Merged columns can be used across files as well as within a single file (important!). Explain implications for suppliers and parts. "As a matter of fact, given that TR allows us to include values in the Field Values Table that don't actually appear at this time in any relation in the database, we might regard TR as a true *domain-oriented* representation of the entire database!"

A.6 Implementing the Relational Operators

Self-explanatory (but important!). The remarks about *symmetric exploitation* and *symmetric performance* are worth some attention. *Note:* The same is true for the unanswered questions at the end of the summary section (fire students up to find out more for themselves!).

Where can I buy one?

***** End of Appendix A *****

Appendix C

S Q L E x p r e s s i o n s

Principal Sections

- Table expressions
- Boolean expressions

General Remarks

This appendix is primarily included for reference purposes. I wouldn't expect detailed coverage of the material in a live class. Also, note the following:

(Begin quote)

[We] deliberately omit:

- Details of scalar expressions
- Details of the RECURSIVE form of WITH
- Nonscalar <*select item*>s
- The ONLY variants of <*table ref*> and <*type spec*>
- The GROUPING SETS, ROLLUP, and CUBE options on GROUP BY
- BETWEEN, OVERLAPS, and SIMILAR conditions
- Everything to do with nulls

We should also explain that the names we use for syntactic categories and SQL language constructs are mostly different from those used in the standard itself [4.23], because in our opinion the standard terms are often not very apt.

(End quote)

Here for your information are a couple of examples of this last point:

- The standard actually uses "qualified identifier" to mean, quite specifically, an identifier that is *not* qualified!

Copyright (c) 2003 C.J.Date

- It also uses "table definition" to refer to what would more accurately be called a "*base* table definition" (the standard's usage here obscures the important fact that a *view* is also a defined table, and hence that "table definition" ought to include "view definition" as a special case).

Actually, neither of these examples is directly relevant to the grammar presented in the book, but they suffice to illustrate the point.

***** End of Appendix B *****

Appendix B

Abbreviations, Acronyms, and Symbols

Like Appendix B, this appendix is primarily included for reference purposes. I wouldn't expect detailed coverage of the material in a live class. However, I'd like to explain the difference between an abbreviation and an acronym, since the terms are often confused. An abbreviation is simply a shortened form of something; e.g., *DBMS* is an abbreviation of *database management system*. An acronym, by contrast, is a *word* that's formed from the initial letters of other words; thus, *DBMS* isn't an acronym, but *ACID* is.* It's true that some abbreviations become treated as words in their own right, sooner or later, and thus become acronyms——e.g., *laser, radar*——but not all abbreviations are acronyms.

———————

* Thus, the well-known "TLA" (= three letter acronym) is *not* an acronym!

———————

***** End of Appendix C *****

Copyright (c) 2003 C.J.Date

Appendix D

Storage Structures and Access Methods

Principal Sections

- Database access: an overview
- Page sets and files
- Indexing
- Hashing
- Pointer chains
- Compression techniques

General Remarks

Personally, I wouldn't include the material of this appendix in a live class (it might make a good reading assignment). In the early days of database management (late 1960s, early 1970s) it made sense to cover it live, because (a) storage structures and access methods were legitimately regarded as part of the subject area, and in any case (b) not too many people were all that familiar with it. Neither of these reasons seems valid today:

a. First, storage structures and access methods have grown into a large field in their own right (see the "References and Bibliography" section in this appendix for evidence in support of this claim). In other words, I think that what used to be regarded as the field of database technology has now split, or should now be split, into two more or less separate fields——the field of database technology as such (the subject of the present book), and the supporting field of *file management*.

b. Second, most students now do have a basic understanding of that file management field. There are certainly college courses and whole textbooks devoted to it. (Regarding the latter, see, e.g., references [D.1], [D.10], and [D.49].)

If you do decide to cover the material in a live class, however, then I leave it to you as to which topics you want to emphasize and which omit (if any). Note that the appendix as a whole is concerned only with traditional techniques (B-trees and the like); Appendix A offers a very different perspective on the subject.

Copyright (c) 2003 C. J. Date

Section D.7 includes the following inline exercise. We're given that the data to be represented involves only the characters A, B, C, D, E, also that those five characters are Huffman-coded as indicated in the following table:

Character	*Code*
E	1
A	01
D	001
C	0001
B	0000

Exercise: What English words do the following strings represent?

00110001010011

010001000110011

Answers: DECADE; ACCEDE.

Answers to Exercises

Note the opening remarks: "Exercises D.1-D.8 might prove suitable as a basis for group discussion; they're intended to lead to a deeper understanding of various physical database design considerations. Exercises D.9 and D.10 have rather a mathematical flavor."

D.1 *No answer provided.*

D.2 *No answer provided.*

D.3 *No answer provided.*

D.4 *No answer provided.*

D.5 The advantages of indexes include the following:

- They speed up direct access based on a given value for the indexed field or field combination. Without the index, a sequential scan would be required.

- They speed up sequential access based on the indexed field or field combination. Without the index, a sort would be required.

The disadvantages include:

- They take up space on the disk. The space taken up by indexes can easily exceed that taken up by the data itself in a heavily indexed database.

- While an index will probably speed up retrieval operations, it will at the same time slow down update operations. Any INSERT or DELETE on the indexed file or UPDATE on the indexed field or field combination will require an accompanying update to the index.

See the body of the chapter and Appendix A for further discussion of the advantages and disadvantages, respectively.

D.6 In order to maintain the desired clustering, the DBMS needs to be able to determine the appropriate physical insert point for a new supplier record. This requirement is basically the same as the requirement to be able to locate a particular record given a value for the clustering field. In other words, the DBMS needs an appropriate access structure——for example, an index——based on values of the clustering field. *Note:* An index that's used in this way to help maintain physical clustering is sometimes called a **clustering index**. A given file can have at most one clustering index, by definition.

D.7 Let the hash function be *h,* and suppose we wish to retrieve the record with hash field value *k*.

- One obvious problem is that it isn't immediately clear whether the record stored at hash address *h*(*k*) is the desired record or is instead a collision record that has overflowed from some earlier hash address. Of course, this question can easily be resolved by inspecting the value of the hash field in the record in question.

- Another problem is that, for any given value of *h*(*k*), we need to be able to determine when to stop the process of sequentially searching for any given record. This problem can be solved by keeping an appropriate flag in the record prefix.

- Third, as pointed out in the introduction to the subsection on extendable hashing, when the file gets close to full, it's likely that most records won't be stored at their hash address location but will instead have overflowed to some other position. If record *r1* overflows and is therefore stored at hash address *h2,* a record *r2* that subsequently hashes to *h2* might be forced to overflow to *h3*——even though there might as

yet be no records that actually hash to *h2* as such. In other words, the collision-handling technique itself can lead to further collisions. As a result, the average access time will go up, perhaps considerably.

D.8 This exercise is answered, in part, in Section D.6.

D.9 (a) 3. (b) 6. For example, if the four fields are *A*, *B*, *C*, *D*, and if we use the appropriate ordered combination of field names to denote the corresponding index, the following indexes will suffice: *ABCD*, *BCDA*, *CDAB*, *DABC*, *ACBD*, *BDAC*. (c) In general, the number of indexes required is equal to the number of ways of selecting n elements from a set of N elements, where n is the smallest integer greater than or equal to $N/2$——i.e., the number is $N! / (n! * (N-n)!)$. For proof see Lum [D.21].

D.10 The number *levels* in the B-tree is the unique positive integer k such that $n^{k-1} < N \leq n^k$. Taking logs to base n, we have $k - 1 < \log_n N \leq k$

$$k = \text{ceil}(\log_n N),$$

where ceil(x) denotes the smallest integer greater than or equal to x.

Now let the number of pages in the ith level of the index be P_i (where $i = 1$ corresponds to the lowest level). We show that

$$P_i = \text{ceil}\left(\frac{N}{n^i}\right)$$

and hence that the total number of pages is

$$\sum_{i=1}^{i=k} \text{ceil}\left(\frac{N}{n^i}\right)$$

Consider the expression

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = x, \text{ say.}$$

Suppose $N = qn^i + r (0 \le r \le n^i - 1)$. Then

(a) If $r = 0$,

$$x = \text{ceil}\left(\frac{q}{n}\right) = \text{ceil}\left(\frac{qn^i}{n^{i+1}}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right)$$

(b) If $r > 0$, $\quad x = \text{ceil}\left(\frac{q + 1}{n}\right)$

Suppose $q = q'n + r' (0 \le r' \le n - 1)$. Then
$N = (q'n + r')n^i + r = q'n^{i+1} + (r'n^{i+1} + r)$; since $0 < r \le n^i - 1$ and
$0 \le r^i \le n - 1$,

$$0 < (r'n^i + r) \le n^{i+1} - (n^i - n^{i+1}) < n^{i+1}$$

hence $\text{ceil}\left(\frac{N}{n^{i+1}}\right) = q' + 1$.
But

$$x = \text{ceil}\left(\frac{q'n + r' + 1}{n}\right) = q' + 1$$

since $1 \le r' + 1 \le n$. Thus in both cases (a) and (b) we have that

$$\text{ceil}\left(\frac{\text{ceil}\left(\frac{N}{n^i}\right)}{n}\right) = \text{ceil}\left(\frac{N}{n^{i+1}}\right)$$

Now, it is immediate that $P_1 = \text{ceil}(N/n)$. It is also immediate that $P_1 + 1 = \text{ceil}(P_i / n)$, $1 \le i \le k$. Thus, if $P_i = \text{ceil}(N/n^i)$, then

$$P_{i+1} = ceil\left(\frac{ceil\left(\frac{N}{n^i}\right)}{n}\right) = ceil\left(\frac{N}{n^{i+1}}\right)$$

The rest follows by induction.

Copyright (c) 2003 C. J. Date

D.11 *Values recorded in index* *Expanded form*

```
0 - 2 - Ab                  Ab
1 - 3 - cke                 Acke
3 - 1 - r                   Ackr
1 - 7 - dams,T+             Adams,T+
7 - 1 - R                   Adams,TR
5 - 1 - o                   Adamso
1 - 1 - l                   Al
1 - 1 - y                   Ay
0 - 7 - Bailey,             Bailey,
6 - 1 - m                   Baileym
```

Points arising:

1. The two figures preceding each recorded value represent, respectively, the number of leading characters that are the same as those in the preceding value and the number of characters actually stored.

2. The expanded form of each value shows what can be deduced from the index alone (via a sequential scan) without looking at the indexed records.

3. The "+" characters in the fourth line represent blanks.

4. We assume the next value of the indexed field doesn't have "Baileym" as its first seven characters.

The percentage saving in storage space is 100 * (150 - 35) / 150 percent = 76.67 percent.

The index search algorithm is as follows. Let *V* be the specified value (padded with blanks if necessary to make it 15 characters long). Then:

```
found := false ;
do for each index entry in turn ;
   expand current index entry and let expanded length = N ;
   if expanded entry = leftmost N characters of V
   then do ;
           retrieve corresponding record ;
           if value in that record = V
           then found := true ;
           leave loop ;
        end ;
   if expanded entry > leftmost N characters of V
   then leave loop ;
end ;
if found = false
then /* no record for V exists */ ;
```

```
    else /* record for V has been found */ ;
```

For "Ackroyd,S" we get a match on the third iteration; we retrieve the corresponding record and find that it is indeed the one we want.

For "Adams,V" we get "index entry high" on the sixth iteration, so no corresponding record exists.

For "Allingham,M" we get a match on the seventh iteration; however, the record retrieved is for "Allen,S", so it's permissible to insert a new one for "Allingham,M". (We're assuming here that indexed field values are required to be unique.) Inserting "Allingham,M" involves the following steps.

1. Finding space and storing the new record

2. Adjusting the index entry for "Allen,S" to read

   1 - 3 - lle

3. Inserting an index entry between those for "Allen,S" and "Ayres,ST" to read

   3 - 1 - i

Note that the preceding index entry has to be changed. In general, inserting a new entry into the index can affect the preceding entry or the following entry, or possibly neither——but never both.

***** End of Appendix D *****

Copyright (c) 2003 C. J. Date